

TURING

图灵程序设计丛书

PRENTICE
HALL

Embedded Linux Primer

嵌入式Linux基础教程

[美] Christopher Hallinan 著
华清远见嵌入式培训中心 译

- 嵌入式Linux权威著作
- Amazon全五星评价
- 全面剖析嵌入式Linux开发，揭示大量技术内幕



人民邮电出版社
POSTS & TELECOM PRESS

“这本书很令我振奋，它为那些想在嵌入式系统中使用Linux的开发人员提供了极好的学习路线指导。本书内容简洁、准确，组织合理，Christopher的知识和见解贯穿全书，你不仅能得到很多信息和帮助，也能享受到阅读的乐趣。”

——Arnold Robbins，著名Linux专家

“本书涵盖了嵌入式Linux开发的方方面面……强烈推荐每一位嵌入式Linux开发人员阅读。”

——LinuxQuestions.org

Embedded Linux Primer

嵌入式Linux基础教程

广泛的硬件支持、高效稳定的内核、开源共享的软件、优秀的开发工具、完善的网络通信和文件管理机制等特点，使嵌入式Linux获得了广泛应用，已成为嵌入式开发的主流平台。

本书是嵌入式Linux领域名著，全面深入而又简明地阐述了构建嵌入式Linux系统的精髓。书中不仅剖析了嵌入式Linux系统，而且描述了处理器、内核、引导装入程序、设备驱动程序、文件系统等关键组件，介绍了嵌入式Linux系统的开发工具和调试技术。书中作者多年积累总结的嵌入式Linux开发技巧和提示，无论对初学者还是有经验的开发人员，都弥足珍贵。

译者特别提供了本书内容的答疑服务，网址为<http://www.farsight.com.cn/FarsightBBS/index.asp>。



Christopher Hallinan 著名嵌入式Linux技术专家，现任Monta Vista软件公司现场应用工程师，曾任3Com公司工程总监。他有25年以上网络和通信产品的软硬件开发经验，曾担任Linux咨询师，提供定制Linux主板接口、设备驱动程序和引导装入程序等方面的解决方案。

翻译团队

华清远见嵌入式培训中心（<http://www.farsight.com.cn>）是享有盛誉的嵌入式高端培训企业，目前已成为ARM、Altera、Atmel、Microsoft、Symbian等全球知名嵌入式企业授权培训中心，每年为Samsung、NEC、Philips、Motorola等世界500强企业提供嵌入式技术企业培训服务，同时也致力于推广与普及嵌入式技术，数万名技术人员由此受益。



www.informit.com

本书相关信息请访问：图灵网站 <http://www.turingbook.com>

读者/作者热线：(010)51095186

反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机/程序设计/Linux

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-21522-2



9 787115 215222 >

ISBN 978-7-115-21522-2

定价：59.00元

译者序

如果早些看到 Christopher Hallinan 的这本书，我就不会在开发过程中走那么多弯路了！

人类无限膨胀的欲望促进了嵌入式的发展。回想 10 年前，你能想象手机可以照相吗？你能想象汽车里会安装全球定位系统吗？今天看来，这些都是多么普通的功能，因为只要调用一些函数（压缩算法）就可以实现。但是你想过吗？如果没有操作系统的支持，很多复杂的功能是无法完成的。要从事嵌入式开发，掌握操作系统的知识是必要的本领之一。

一个不可忽视的事实是电子产品的性能不断提升，而价格却在下降。开发商越来越重视成本。免费、自由的 Linux 无疑是一个强有力的竞争者。凭借优异的特性和良好的发展趋势，Linux 轻而易举地坐上了嵌入式操作系统的头把交椅。

嵌入式系统并不见得有多么高深，但是因为嵌入式系统本身涉及了很多学科，致使很多初学者时常深感迷茫，不知道从何入手，即便是编译环境都很难搭建，更不用说调试和部署了。幸运的是，Christopher Hallinan 的这部著作作为我们学习嵌入式系统提供了捷径。请允许我在此使用“捷径”一词，因为我在刚刚踏入嵌入式 Linux 大门时，其中的很多概念也一度让我觉得神秘而困惑。虽然网络搜索功能很强大，但是就如同迷失在一棵大树的树叶之间，你很难摸索到树干，找到正确的方向，而 Christopher Hallinan 的这本书就是指引我们前进的“树干”。更可贵的是，本书每章后都提供了相关参考资料，你会很容易地查找到需要了解的内容。

在如此短的篇幅内阐述嵌入式 Linux 的方方面面是不可能的。但本书作者却让你在一本书中轻松地掌握了嵌入式开发的脉络，这是难能可贵的。本书内容广泛而又不乏深度，嵌入式 Linux 开发的初学者和提高者都能从中获得巨大收获。

本书的翻译工作由北京华清远见科技信息有限公司负责组织，拿到书后，我们的翻译团队粗略地浏览了一遍，一致认为本书的内容尽在我们的掌握之中，毕竟我们自认为在嵌入式 Linux 领域小有经验。但在翻译过程中，我们渐渐改变了最初的错误的想法。看起来和写出来有很大不同。除了赞叹作者扎实的基本功外，我们更被作者高超的写作艺术深深折服。摆在我们面前最大的难题不是技术，而是如何尽可能地把作者的原意表现出来。我想，这也是衡量一本书翻译质量的关键吧！

翻译的具体分工如下：王辉翻译第 1 章至第 4 章、第 8 章，张小全翻译第 5 章、第 6 章、第 9 章至第 11 章，其余部分及全书统稿由孙天泽完成。我要特别感谢袁文菊、吴彦波两位老师对本书所做的贡献。

我希望能够代表嵌入式同行们感谢人民邮电出版社图灵公司，是他们以卓越的眼光引进了这部著作。尽管我们做了充分的准备，但是受能力所限，译文中仍难免存在一些错误，还请读者批评指正。最后，祝读者能通过学习本书获得较大的提高。

序

计算机无处不在！

在过去大约 25 年中，只要不是与世隔绝的人就肯定不会对此感到大惊小怪。现在，计算机不仅占据了我们的桌面，进驻了我们的厨房，而且越来越多地进入到我们的生活场所，即便是在微波炉、电烤箱、移动电话和便携式数字音乐播放器中也出现了它的身影。

选择本书的读者肯定已经了解了不少，但还想学习更多的嵌入式系统知识。

就在不久前，嵌入式系统还不是很强大，它们运行具有特殊目的、专用的操作系统，而这些操作系统与工业标准的系统有很大不同（而且，它们也更难于开发）。现在，嵌入式系统即使在功能上不比家用计算机强大，但至少也与其相当（例如高端游戏终端）。

伴随着这种强大的功能，运行 Linux 等成熟操作系统的能力也呼之欲出，在嵌入式产品中使用 Linux 这样的操作系统变得具有非常大的意义。一个庞大的开发者社区更使得这一切成为可能。开发环境和部署环境惊人相似，这也使得程序员的生活变得更轻松。现在我们既有由虚拟内存系统提供的保护地址空间的安全性，又有多用户的能力和灵活性。真是不老少了。

出于这个原因，全世界的公司都在许多设备中选择使用 Linux，如 PDA、家庭娱乐系统，甚至移动电话——不管你信不信！

这本书很令我振奋。它为那些想在嵌入式系统中使用 Linux 的开发人员提供了极好的学习路线指导。本书内容简洁、准确，组织合理，Christopher 的知识和见解贯穿全书，你不仅能得到很多信息和帮助，也能获得阅读的乐趣。

我希望在你学习的同时也能感受到这种乐趣，我自己已经感受到了。

Arnold Robbins

（著名 Linux 专家）



前言

虽然 Linux 方面已经有很多好书，但是没有哪一本书能为嵌入式 Linux 开发人员提供广泛的信息和建议。当然，有一些非常优秀的书籍介绍了 Linux 内核和 Linux 系统管理等方面的知识，本书也参考了许多我认为在同类书中最优秀的著作。

本书的大部分素材取自我在这些年来收到的一些开发工程师提出的问题，当时我的职位是嵌入式 Linux 顾问。现在我是 Monta Vista Software 公司的现场应用工程师，该公司是嵌入式 Linux 发行厂商的领跑者。

即便对于很有经验的软件工程师来说，嵌入式 Linux 也带来了一些特殊的挑战。首先，那些具有多年实时操作系统（RTOS）开发经验的工程师很难把思维转换到 Linux 上；其次，有经验的应用程序开发人员通常很难理解多种开发环境的复杂性。

虽然这只是一本面向刚接触嵌入式 Linux 开发人员的基础读物，但是我确信有经验的嵌入式 Linux 开发人员也一定能从中找到有用的提示和技巧，这些可是我花费多年积累总结出来的。

给嵌入式 Linux 开发者的实用建议

书中包括了我的一些观点。作为一名嵌入式工程师，要跟上嵌入式 Linux 环境的快速发展，你需要知道这些观点。本书没有重点讲解 Linux 内核内部原理，在谈论内核的章节中侧重从项目角度介绍内核，你可以阅读专门介绍内核内部原理的著作来了解相关知识。通过本书可以学习内核源码树的组织和布局，了解组成内核映像的二进制文件组件以及如何加载它们，它们在嵌入式系统中的作用等知识。图 5-1 是我最欣赏的一幅图，它形象地说明了合成内核映像的构建过程。

本书的一些章节讲述了构建系统的工作原理，以及怎样将满足项目需求的定制的内核变化加载到内核中。你会了解用于驱动不同体系结构配置的机制和 Linux 内核源码树的特性；更重要的是，掌握如何修改系统使之满足自己的需求。除此之外，我们还深入探讨了内核命令行参数机制，介绍了它是如何工作的，如何根据需求配置内核运行时行为，如何扩展系统功能，如何导航内核源代码，如何为相关嵌入式系统的不同任务配置内核。其他内容还包括嵌入式项目中一些非常实用的提示和技巧，内容涵盖了引导装入程序、系统初始化、文件系统和闪存、内核调试技巧以及应用程序调试技巧等。

读者对象

本书需要读者具有一定的 C 语言编程基础，对局域网和因特网有基本的了解，理解 IP 地址的概念以及 IP 地址在简单局域网中的用法，还需要理解十六进制和八进制编码方式以及它们常见的用法。

本书也涉及一些 C 语言编译和链接中较为深入的概念，所以如果你能粗略复习一下 C 语言链接器的概念就更好了。同时，了解 GNU make 操作和语法对于阅读本书也很有帮助。

本书不是什么

本书不是一本详细介绍硬件的指南。嵌入式开发者所面临的困难之一就是现在硬件设备之间有巨大的差异。一款集成部分外围设备的现代 32 位处理器，其用户手册动辄就有 1000 页，这没有捷径可走。但从程序员的角度看，如果需要理解硬件设备，你必须花费大量时间研读硬件数据手册和参考指南，同时要花费更多的时间编写和测试这些硬件设备的工作代码。

这也不是一本讲述 Linux 内核和内部原理的书。从本书中无法学到用来实现 Linux 虚拟内存管理策略和过程的内存管理单元（MMU）的精深知识。已经有许多关于这个主题的优秀书籍，我建议你翻阅每章后面的“参考资源”。

排版约定

文件名和代码采用 Courier 字体，需要读者输入的命令使用加粗 **Courier** 字体。新术语或重要的概念使用楷体加以强调。

路径名前如有 3 个点则表示众所周知但未明确指定的顶层目录。上下文不同，顶层目录也会不同，但大多数情况下是指 Linux 内核源码目录的顶层。例如，`.../arch/ppc/kernel/setup.c` 表示 `setup.c` 文件位于 Linux 内核源码树的体系结构分支上。实际路径可能是 `~/sandbox/linux.2.6.14/arch/ppc/kernel/setup.c`。

本书结构

第 1 章简要介绍了 Linux 被迅速应用在嵌入式环境的驱动因素，介绍了与嵌入式 Linux 相关的几个重要的标准和组织。

第 2 章介绍了许多与后几章所构建的嵌入式 Linux 相关的概念。

第 3 章将站在更高的层面了解用于构建嵌入式 Linux 系统的流行的处理器和平台，介绍了从主要处理器厂商精选的几款产品，以及几乎所有主流的体系结构。

第 4 章从略微不同的角度审视 Linux 内核。这里没有重点讲解内核理论或其内部原理，只是介绍了内核的结构、布局和构建结构，目的是使读者从一开始就能学习这门庞大的软件工程项目。更重要的是，要知道哪些内容是必须重点关注的，包括对内核构建系统的详细讲解。

第 5 章详细说明了 Linux 内核的初始化过程。你可以学习到与体系结构和引导装入程序相关

的映射组件，是如何拼接成适合下载到闪存的内核映射，并最终通过嵌入式系统的引导装入程序启动的。从这一章学到的知识将帮助你自定义 Linux 内核，使之可以满足你自己的嵌入式应用的需求。

第 6 章继续讲述初始化过程。当 Linux 内核完成自身初始化后，应用程序将根据预先确定的方式继续初始化过程。读完这一章以后，你就具备了自定义用户空间应用程序启动顺序的知识。

第 7 章主要介绍引导装入程序及其在嵌入式 Linux 系统中的作用。这一章以现在流行的开源引导装入程序 U-Boot 为例，说明了移植的概念；还简要介绍了其他几种现在使用着的引导装入程序，以使用户有特殊需求时可以有多种选择。

第 8 章介绍了 Linux 设备驱动程序模型，提供了很多进行设备驱动程序开发的背景资料，这些资料都在“参考资源”中列出。

第 9 章列举了目前嵌入式系统中使用的一些流行的文件系统，包括在闪存设备上最常用的 JFFS2 文件系统。这一章还简要介绍了如何创建自己的文件系统映像，这也是嵌入式 Linux 开发人员所面临的一项艰巨任务。

第 10 章介绍了 MTD (Memory Technology Devices, 内存技术设备) 子系统。MTD 是 Linux 文件系统和硬件内存设备 (尤其是闪存) 之间一种非常有效的抽象层。

第 11 章介绍了 BusyBox，它是我们构建小型嵌入式系统最常用的工具。这一章讲述如何根据特殊需求对 BusyBox 进行配置和构建，随后介绍了仅使用 BusyBox 环境完成系统初始化的全过程。附录 B 列举了最新版本 BusyBox 提供的命令。

第 12 章详细介绍了典型交叉开发环境的特殊需求。这一章所介绍的一些技术能有效地提高嵌入式开发人员的工作效率，例如强大的 NFS 根目录挂载开发配置。

第 13 章介绍了一些有用的开发工具。介绍了使用 gdb 进行调试，包括核心转储分析；并通过示例介绍了 strace、ltrace、top 和 ps，以及内存剖析工具 mtrace 和 dmalloc。这一章最后介绍了更重要的一些二进制实用工具，如 readelf 等。

第 14 章深入探讨了一些 Linux 内核的调试技术，介绍了内核调试器 KGDB 的用法，提出了 gdb 和 KGDB 组合使用的许多调试技巧。这一章涉及的内容还包括硬件 JTAG 调试器的用法，以及当内核无法启动时的一些故障分析技巧。

第 15 章把调试环境从内核转移至应用程序。这一章继续完善前两章用到的 gdb 示例，讲述了多线程和多进程的调试技巧。

第 16 章介绍了将 Linux 移植到自定义开发板的相关问题。这一章通过一个简单的示例，逐步说明了 Linux 内核移植到 PowerPC 板的详细过程，还讲解了几个困扰 Linux 内核移植方面新手的重要概念。读完本章后，会同第 13 章和第 14 章提出的技术，你应该能够对自己的开发板进行移植工作。

第 17 章介绍了嵌入式 Linux 中一个令人激动的发展：通过配置 CONFIG_RT 选项实现实时。这里介绍的特性通过 RT 选项得以实现，同时还介绍了如何在设计中使用这些特性。这一章也介绍了在应用程序中测试延时的技巧。

附录内容包括 U-Boot 可配置命令、BusyBox 命令、SDRAM 接口的注意事项、开源开发者

的资源、BDI-2000 调试器的配置文件范例。BDI-2000 是目前很流行的硬件 JTAG 调试器。

其他

如果你能够边看书边在 Linux 工作站上动手实验，将会从书中得到最大的收获。可以找一个较旧的 x86 计算机完成嵌入式系统实验。如果有条件能连接其他体系结构的平台进行实验就更好了。你将受益于学习到大型代码库（如 Linux 内核）的布局和组织，在浏览内核并亲自动手实验时，能学到一些更重要的知识和经验。

看一下本书使用的代码并试着理解书中的示例，要使用不同的设置方案、配置选项和不同的硬件设备进行实验。除可获得丰富的知识，还充满了乐趣！

版权说明

本书使用的开源代码的版权归很多个人或公司所有。复制代码遵循了 GNU 公共许可，即 GPL。

致谢

我由衷地敬佩开源软件工程师的崇高精神，深深地折服于我们社区中远远超过我的天才们。在本书的创作过程中，我向 Linux 和开源社区的很多人提出了大量问题，大多数问题都能很快得到答案，而且还经常获得鼓励。我要向 Linux 和开源社区中帮我解答问题的朋友致以真挚的谢意（排名不分先后）：

Dan Malek 为第 2 章的部分内容提供了创作灵感。

Dan Kegel 和 Daniel Jacobowitz 耐心地帮我解答了关于工具链的问题。

Scott Anderson 提供了第 14 章中 gdb 宏的最初的思想。

Brad Dixon 不断地用他所掌握的知识挑战和扩展我的技术洞察力。

George Davis 帮我解答了 ARM 的问题。

Jim Lewis 为我提供了关于 MTD 的意见和建议。

Cal Erickson 帮我解答了关于 gdb 用法的问题。

John Twomey 就第 3 章内容给出了建议。

Lee Revell、Sven-Thorsten Dietrich 和 Daniel Walker 就实时 Linux 的内容提供了建议。

非常感谢 AMCC、Embedded Planet、Ultimate Solutions 和 United Electronic Industries 公司，它们提供了示例硬件。感谢我的公司 Monta Vista，允许我进行这次与工作无关的创作，并且提供了一些软件示例。在创作过程中，还有很多人贡献了他们的想法，并给予我鼓励和支持，我也非常感激！

我要诚挚地感谢最初审阅本书的团队，他们迅速地阅读了每一章，提供了极好的反馈、注释和想法。谢谢 Arnold Robbins、Sandy Terrace、Kurt Lloyd 和 Rob Farber。还要感谢 Arnold 帮助我这个写作新手学习撰写技术图书的规则。虽然我已经努力排除每处错误，但错误肯定还会存在，

这都归咎于我。

感谢 Mark L. Taub 使本书得以完成，感谢他的鼓励和无限的耐心。还要感谢制作团队，包括 Kristy Hart、Jennifer Cramer、Krista Hansing 和 Cheryl Lenser。

最后，还要把最特别、最衷心的感谢献给 Cary Dillman，在我撰写本书时她阅读了每一章，整个创作过程中都有她的不断鼓励和重要的贡献。

Christopher Hallinan



目 录

第1章 引言.....1	参考资源.....22
1.1 为什么使用 Linux.....1	第3章 处理器基础.....23
1.2 嵌入式 Linux 现状.....2	3.1 单机处理器.....23
1.3 开源和 GPL.....2	3.1.1 IBM 970FX.....24
1.4 标准和相关机构.....3	3.1.2 Intel Pentium M.....24
1.4.1 LSB.....3	3.1.3 Freescale MPC7448.....25
1.4.2 OSDL.....3	3.1.4 配套芯片组.....25
1.5 小结.....4	3.2 集成化处理器：片上系统.....27
参考资源.....4	3.2.1 PowerPC.....27
第2章 嵌入式初体验.....5	3.2.2 AMCC PowerPC.....27
2.1 需要嵌入式系统吗.....5	3.2.3 Freescale PowerPC.....30
2.2 嵌入式系统剖析.....6	3.2.4 MIPS.....33
2.2.1 典型嵌入式 Linux 系统设置.....7	3.2.5 Broadcom MIPS.....33
2.2.2 启动目标板.....8	3.2.6 AMD MIPS.....34
2.2.3 启动内核.....9	3.2.7 其他类型的 MIPS.....35
2.2.4 内核初始化概述.....10	3.2.8 ARM.....35
2.2.5 第一个用户空间进程：init.....11	3.2.9 TI ARM.....35
2.3 存储的思考.....12	3.2.10 Freescale ARM.....37
2.3.1 闪存.....12	3.2.11 Intel ARM XScale.....37
2.3.2 NAND 闪存.....13	3.2.12 其他 ARM.....38
2.3.3 闪存的用途.....14	3.2.13 其他体系结构.....38
2.3.4 闪存文件系统.....14	3.3 硬件平台.....38
2.3.5 存储器空间.....15	3.3.1 CompactPCI.....38
2.3.6 运行上下文.....16	3.3.2 ATCA.....39
2.3.7 进程中的虚拟内存.....17	3.4 小结.....39
2.3.8 交叉开发环境.....19	参考资源.....40
2.4 嵌入式 Linux 的发行版.....20	第4章 Linux 内核——不同视角.....41
2.4.1 Linux 商业发行版.....21	4.1 背景知识.....41
2.4.2 Linux 自定义发行版.....21	4.1.1 内核的版本.....42
2.5 小结.....21	4.1.2 内核源码库.....43

4.2 Linux 内核构造	44	6.1.4 根文件系统带来的挑战	89
4.2.1 顶层资源目录	44	6.1.5 试错法	90
4.2.2 编译内核	45	6.1.6 自动化文件系统构建工具	90
4.2.3 严格意义上的内核: vmlinux	46	6.2 内核的最后引导过程	90
4.2.4 内核映像组件	47	6.2.1 用户空间下第一个程序	91
4.2.5 子目录结构	50	6.2.2 解决依赖	92
4.3 内核构建系统	50	6.2.3 定制初始化进程	92
4.3.1 .config 文件	51	6.3 init 进程	92
4.3.2 配置编辑器	52	6.3.1 inittab	95
4.3.3 makefile 的目标	55	6.3.2 Web 服务器启动脚本示例	96
4.3.4 内核配置	58	6.4 初始 RAM 磁盘	97
4.3.5 自定义配置选项	59	6.4.1 初始 RAM 磁盘的目的	98
4.3.6 内核 makefile	62	6.4.2 使用 initrd 引导	98
4.3.7 内核文档	62	6.4.3 引导装入程序对于 initrd 的支持	98
4.4 获取 Linux 内核	63	6.4.4 initrd 的奥妙所在: linuxrc 文件	100
4.5 小结	64	6.4.5 initrd 探究	100
参考资源	64	6.4.6 构建 initrd 映像文件	101
第 5 章 内核初始化	65	6.5 使用 initramfs	102
5.1 合成内核映像: piggy 及其他	65	6.6 关机	103
5.1.1 Image 目标文件	67	6.7 小结	103
5.1.2 体系结构相关的目标文件	68	参考资源	104
5.1.3 第二阶段引导装入程序	69	第 7 章 引导装入程序	105
5.1.4 引导信息	69	7.1 引导装入程序的作用	105
5.2 初始化控制流	72	7.2 引导装入程序的挑战	106
5.2.1 内核入口点: head.o	73	7.2.1 DRAM 控制器	106
5.2.2 内核启动: main.c	74	7.2.2 闪存与 RAM	106
5.2.3 体系结构设置	75	7.2.3 映像的复杂性	107
5.3 内核命令行处理	75	7.2.4 执行上下文	108
5.4 子系统初始化	80	7.3 通用的引导装入程序: Das U-Boot	109
5.5 init 线程	82	7.3.1 执行上下文	109
5.5.1 通过 initcall 初始化	83	7.3.2 U-Boot 命令集	111
5.5.2 引导的最后步骤	84	7.3.3 网络操作	111
5.6 小结	85	7.3.4 存储子系统	113
参考资源	85	7.3.5 从磁盘启动: U-Boot	113
第 6 章 系统初始化	86	7.4 移植 U-Boot	114
6.1 根文件系统	86	7.4.1 为 EP405 开发板移植 U-Boot	114
6.1.1 FHS	87		
6.1.2 文件系统布局	87		
6.1.3 最小文件系统	88		

7.4.2 U-Boot 的 makefile 配置目标	115	9.3 ext3 文件系统	150
7.4.3 EP405 处理器初始化	116	9.4 ReiserFS 文件系统	152
7.4.4 特定开发板的初始化	117	9.5 JFFS2 文件系统	153
7.4.5 移植概要	120	9.6 cramfs 文件系统	155
7.4.6 U-Boot 映像格式	120	9.7 NFS 文件系统	156
7.5 其他引导装入程序	122	9.8 伪文件系统	160
7.5.1 Lilo	122	9.8.1 proc 文件系统	160
7.5.2 GRUB	123	9.8.2 sysfs 文件系统	162
7.5.3 其他引导装入程序	124	9.9 其他文件系统	164
7.6 小结	124	9.10 构建简单的文件系统	165
参考资源	124	9.11 小结	166
第 8 章 设备驱动程序基础	126	参考资源	166
8.1 设备驱动程序基本概念	126	第 10 章 MTD 子系统	168
8.1.1 可加载模块	127	10.1 启用 MTD 服务	168
8.1.2 设备驱动程序的体系结构	127	10.2 MTD 基础知识	170
8.1.3 最小设备驱动程序示例	128	10.3 MTD 分区	172
8.1.4 模块构建的基础设施	129	10.3.1 Redboot 分区表	173
8.1.5 安装设备驱动程序	131	10.3.2 内核命令行分区	176
8.1.6 加载设备驱动程序模块	132	10.3.3 映射驱动程序	177
8.2 模块实用程序	133	10.3.4 闪存芯片驱动程序	178
8.2.1 insmod	133	10.3.5 特定开发板的初始化	179
8.2.2 模块参数	133	10.4 MTD 实用程序	180
8.2.3 lsmod	134	10.5 小结	184
8.2.4 modprobe	135	参考资源	184
8.2.5 depmod	136	第 11 章 BusyBox	186
8.2.6 rmmmod	136	11.1 BusyBox 简介	186
8.2.7 modinfo	137	11.2 BusyBox 配置	187
8.3 驱动程序方法	137	11.3 BusyBox 操作	189
8.3.1 驱动程序文件系统操作	138	11.3.1 BusyBox 之 init	191
8.3.2 设备节点与 mknod	140	11.3.2 rcS 初始化脚本示例	193
8.4 汇总	141	11.3.3 在目标平台安装 BusyBox	193
8.5 设备驱动程序与 GPL	143	11.3.4 BusyBox 命令	195
8.6 小结	143	11.4 小结	196
参考资源	144	参考资源	196
第 9 章 文件系统	145	第 12 章 嵌入式开发环境	197
9.1 Linux 文件系统的概念	146	12.1 交叉开发环境	197
9.2 ext2 文件系统	147	12.2 主机系统需求	200
9.2.1 挂载文件系统	148	12.3 为目标板提供服务	201
9.2.2 文件系统完整性检查	149		

12.3.1	TFTP 服务器	201	第 14 章	内核调试技术	238
12.3.2	BOOTP/DHCP 服务器	202	14.1	内核调试的难点	238
12.3.3	NFS 服务器	204	14.2	使用 KGDB 调试内核	239
12.3.4	使用 NFS 为目标板挂载根文件系统	205	14.2.1	KGDB 内核配置	240
12.3.5	U-Boot NFS 根挂载示例	206	14.2.2	支持 KGDB 的内核启动	241
12.4	小结	208	14.2.3	有用的内核断点	243
	参考资源	208	14.3	Linux 内核的调试	244
第 13 章	开发工具	209	14.3.1	gdb 远程串口协议	244
13.1	GDB	209	14.3.2	调试优化后的内核代码	247
13.1.1	调试核心转储	210	14.3.3	gdb 用户定义命令	251
13.1.2	调用 GDB	211	14.3.4	有用的内核 gdb 宏	252
13.1.3	GDB 调试会话	213	14.3.5	调试可加载模块	258
13.2	DDD	214	14.3.6	printk 调试	262
13.3	cbrowser/csopce	216	14.3.7	Magic SysReq 键	263
13.4	追踪和程序分析工具	217	14.4	硬件辅助调试	263
13.4.1	strace	217	14.4.1	使用 JTAG 探测器对闪存编程	265
13.4.2	strace 的变体	220	14.4.2	用 JTAG 探测器进行调试	266
13.4.3	ltrace	221	14.5	无法启动时	268
13.4.4	ps	222	14.5.1	早期串口调试输出	269
13.4.5	top	224	14.5.2	转储 printk 日志缓冲区	270
13.4.6	mtrace	225	14.5.3	KGDB 捕捉崩溃	271
13.4.7	dmalloc	226	14.6	小结	272
13.4.8	内核 oops	228		参考资源	272
13.5	二进制实用程序	230	第 15 章	调试嵌入式 Linux 应用程序	274
13.5.1	readelf	230	15.1	目标机调试	274
13.5.2	使用 readelf 检查调试信息	232	15.2	远程(交叉)调试	274
13.5.3	objdump	233	15.3	使用共享库进行调试	278
13.5.4	objcopy	234	15.4	多任务调试	282
13.6	其他二进制实用程序	234	15.4.1	多进程的调试	282
13.6.1	strip	234	15.4.2	多线程应用程序的调试	284
13.6.2	addr2line	235	15.4.3	引导装入程序/闪存代码的调试	286
13.6.3	strings	235	15.5	远程调试的附加选项	287
13.6.4	ldd	235	15.5.1	串行端口调试	287
13.6.5	nm	236	15.5.2	绑定到正在运行的进程	287
13.6.6	prelink	236	15.6	小结	288
13.7	小结	237		参考资源	288
	参考资源	237			

第 16 章 移植 Linux	289	17.3 实时内核补丁	310
16.1 Linux 源代码的组织	289	17.3.1 实时的特性	311
16.2 为开发板定制 Linux	291	17.3.2 O(1) 调度器	313
16.2.1 前提和假设	291	17.3.3 创建实时进程	313
16.2.2 定制内核初始化	292	17.3.4 临界区管理	314
16.2.3 静态内核命令行	294	17.4 调试实时内核	314
16.3 平台初始化	295	17.4.1 软锁检测	314
16.3.1 早期变量访问	298	17.4.2 抢占调试	315
16.3.2 开发板信息结构	299	17.4.3 调试唤醒时间	315
16.3.3 机器相关的调用	301	17.4.4 唤醒延迟历史	315
16.4 汇总	302	17.4.5 中断响应时间	316
16.5 小结	304	17.4.6 中断响应历史	316
参考资源	304	17.4.7 延迟跟踪	317
第 17 章 Linux 与实时	305	17.4.8 调试死锁环境	318
17.1 什么是实时	305	17.4.9 锁模式的运行时控制权	319
17.1.1 软实时	305	17.5 小结	319
17.1.2 硬实时	306	参考资源	319
17.1.3 Linux 调度	306	附录 A 可配置的 U-Boot 命令	320
17.1.4 中断延迟	306	附录 B BusyBox 命令	322
17.2 内核抢占	307	附录 C SDRAM 接口的注意事项	328
17.2.1 抢占的缺陷	307	附录 D 开源项目资源	334
17.2.2 抢占模型	308	附录 E BDI-2000 配置文件示例	336
17.2.3 SMP 内核	309		
17.2.4 抢占延迟源	310		



第 1 章

引 言

本章内容

- 为什么使用Linux
- 嵌入式Linux现状
- 开源和GPL
- 标准和相关机构
- 小结

现在使用专有操作系统的人越来越少了，他们转而选择Linux之类的自由操作系统，这一趋势在许多传统嵌入式操作系统公司的高层引起了震动。鉴于诸多显而易见的好处，Linux不再局限于服务器应用这一传统阵地，同时还用在大量其他产品中。这些嵌入式系统（embedded system）的例子包括手机、DVD播放器、电视游戏机、数码相机、网络交换机和无线网络设备。Linux极有可能已经出现在你的家中或汽车里。

1.1 为什么使用 Linux

出于很多经济和技术上的考虑，在嵌入式设备中采用Linux操作系统的厂商越来越多。这个趋势实际上贯穿到各类市场和技术领域。目前，Linux已经广泛应用于嵌入式产品中，如全球范围的公共电话交换网络、全球数据网络、无线移动手持设备以及操作这些网络的设备。Linux也成功地应用在汽车及消费产品中，如游戏与PDA、打印机、企业交换机和路由器以及其他产品。嵌入式Linux的采用率持续上升，谁都无法预料将上升多少。

嵌入式Linux的快速发展有下面几个原因。

- Linux已经逐渐取代传统的专有嵌入式操作系统，成为成熟、高性能且稳定的产品。
- Linux支持众多应用程序和连网协议。
- Linux适应性强，既可在小型消费品设备中使用，又可用于大型、重型、电信级交换机和路由器。
- Linux不需要授权就可以进行部署和应用，而传统的专有嵌入式操作系统往往做不到这一点。
- Linux吸引了全世界无数的开发者，可以及时地支持新型的硬件架构、平台和设备。
- 现在，越来越多的硬件和软件行业的厂商（包括顶级制造商和独立软件开发商）都支持Linux。

基于种种原因，Linux已经越来越多地应用在普通的家用电器中，从高清晰电视到移动手持设备等各种产品。

1.2 嵌入式 Linux 现状

不用对Linux在嵌入式领域爆炸式的发展感到惊奇。其实，阅读本书就说明了嵌入式已经触及你的生活。我们无法估量这个市场到底有多大，因为还有许多公司正在开发他们自己的嵌入式Linux发行版。

由Rich Lehrbaum创立的LinuxDevices.com是一个关于Linux新闻和信息的门户网站，每年都会进行一次有关嵌入式Linux市场的调查。根据最近一次调查报告的数据显示，在每年数千个新项目中，Linux已经占据了操作系统的半壁江山。报告称，大约半数被访者基于Linux进行嵌入式设计，紧随其后的操作系统的使用率只占被访者的1/8，而曾经统领嵌入式市场的商业操作系统的使用率已经下降到不足1/10。即使你有充分的理由怀疑这份调查结果，但嵌入式Linux在当今市场中的统治地位已是不争的事实。

1.3 开源和 GPL

促使我们使用Linux的一个根本原因，是因为它是开源的（open source）。Linux内核遵循GNU GPL条款，这意味着Linux是自由软件^①。实际上，GNU GPL在第二段声明：“在谈及自由软件时，我们所说的自由是指可自由使用，而非价格。”GPL条款简明扼要，其中最重要的核心条款是：

- 终生有效；
- 允许用户免费运行程序；
- 允许用户学习和修改源代码；
- 允许用户发布源代码或其修改版；
- GPL软件的传播对象拥有同等权力。

如果软件是根据GPL许可证发布的，就必须始终遵守该许可证^②。即使代码被大量修改（这种修改在GPL中是被允许和提倡的）修改版本仍必须根据GPL条款进行发布。这就是说，任何用户都可以使用GPL软件以及软件的修改版本（也称为衍生版）。

不论包含任何内容，我们发布软件时不会有任何限制，也无需缴纳任何费用。但这并不意味着供货商在GPL软件上无利可图。对于一个GPL软件，不论它是不是衍生（修改）版本，我们都可以对其进行修改并重新发布。但是修改版的作者必须根据GPL条款规定来发布版本。任何衍生版的发行版本，例如用户自定义版本，也同样必须遵循这个规定。

如果你想回顾一下开源运动发展的传奇历程，可以翻阅本章结尾“参考资源”中Eric S. Raymond的著作。

① 大部分专业开发管理者都认同，虽然可以免费下载Linux，但是在嵌入式平台开发和部署任何操作系统都是有成本的（通常还花费不菲）。

② 如果版权所有人同意，理论上软件可以根据新的授权方式发布，但这种情况不太可能出现。

免费与自由

在关于开源的自由本质的讨论中总是离不开free所代表的两个概念：“自由”和“免费”。（对后者更感兴趣。）GPL许可证为软件定义的是“自由”，它确保你可以自由地使用、研究和修改该软件。使用修改版本代码的用户同样拥有这些权力。绝大多数人都非常理解这个概念。

我们听到的最容易让人误解的是，Linux是“免费软件”。的确，Linux是可以免费获取的。也可以在几分钟内下载一个Linux内核。但是，任何专业的开发管理人员都明白，项目中所需要的软件有相当一部分是需要支付费用的，这包括获取、集成、修改、维护和支持各阶段的费用，再加上获取和维护工具链、库、应用程序和针对特定体系结构配置的交叉开发工具的费用。很快你会发现，将你开发的软件包部署到嵌入式Linux系统中也要花费一定的费用。

1.4 标准和相关机构

Linux仍然会继续占领台式机、企业和嵌入式的市场份额，目前正在出台的新标准和机构为Linux的使用和大众化铺平了道路。本节将会介绍你可能想要熟悉的标准。

1.4.1 LSB

与Linux最相关的标准是LSB（Linux标准基础）。LSB的目的是建立一系列标准，以提高不同Linux版本中应用程序之间的互操作性。目前，LSB涵盖了几个体系结构，包括IA32/64、PowerPC 32、PowerPC 64、AMD64及其他常见的体系结构。该标准已形成了核心组件和特定体系结构组件两个部分。

LSB规定了Linux发行版本的公共特性，包括对象格式、标准库接口、最小命令集和实用工具集及其行为、文件系统布局以及系统初始化，等等。

如果想了解更多关于LSB的知识，请参考本章后面的“参考资源”。

1.4.2 OSDL

OSDL（开源开发实验室）的建立促进了Linux挺进大众市场。根据这份声明，OSDL目前能够为Linux社区提供企业级的测试工具以及其他一些技术支持。更重要的是，OSDL还资助了一些工作组来为3个重要的市场领域制定标准，并参与开发各领域相关的特殊应用。下面将依次介绍这3个领域所带来的创新。

1. OSDL：电信级Linux

全球大型网络和电信设备制造商大部分都是基于嵌入式Linux操作系统来开发或组装电信级设备的，用来满足电信级设备的高可靠性、高可用性和快速适用性。这些供应商在制造产品时充分考虑了信息冗余度、交换性、容错性能、协作性以及实时性等各项指标。

OSDL电信级Linux工作组为电信级设备制定了一套需求规范，当前最新版本的规范覆盖了如下7大功能领域。

- 可用性需求：提供高可用性，包括在线维护、冗余和状态监控。
- 集群性需求：提供冗余性服务，例如集群成员的管理及数据的观测。
- 服务性需求：提供远端的服务及维护，例如SNMP（简单网络管理协议）及对风扇和供电

情况的诊断监测。

- 性能需求：定义运行性能、伸缩性、对称多线程处理、延时等性能指标。
- 标准需求：定义电信级兼容设备所遵循的标准。
- 硬件需求：规定高可用性硬件的管理指标，例如刀片服务器组及硬件管理接口等。
- 安全性需求：提高整个系统的安全性以使系统免受攻击。

2. OSDL：移动Linux计划

在编写本书时，市场上已经出现了为数不少的基于嵌入式Linux的手持移动产品（如手机）。据报道，已经有数以百万计的手持设备中使用了Linux。毋庸置疑，这个趋势还将扩大。这预示着嵌入式Linux将会占有原本由传统的专有实时操作系统占有的最大的市场份额。以上说明，Linux系统已经为进军商用嵌入式产品领域做好了充分的准备。

OSDL资助了名为“移动Linux计划”的工作小组。根据OSDL网站的说明，它的主要目标是促进Linux在下一代手持移动设备及具有语音及数据服务的便携式设备上的应用。这个工作组关注的主要工作领域包括：开发工具、I/O及连网、内存管理、多媒体设备、性能模块、电源管理、安全性及存储模块。

3. OSDL：业务可用性论坛

如果所开发的产品要求高可靠性、高可用性及高维护性（Reliability, Availability, Serviceability, RAS），你应该熟悉业务可用性论坛（SA论坛）。这个组织在电信级设备和其他商业设备的系统管理通用接口的制定中扮演着非常重要的角色。业务可用性论坛的网址为www.saforum.org。

1.5 小结

- Linux在开发人员及嵌入式产品制造商中的使用率将持续增长。
- Linux在嵌入式设备中的应用将持续高速增长。
- 列举了促使Linux在嵌入式市场快速增长的几大因素。
- 介绍了影响嵌入式Linux的几大标准和相关机构。

参考资源

The Cathedral and the Bazaar

Eric S. Raymond

O'Reilly Media, Inc., 2001

Linux Standard Base Project

www.linuxbase.org

OSDL（开源开发实验室）

www.osdl.org



嵌入式初体验

本章内容

- 需要嵌入式系统吗
- 嵌入式系统剖析
- 存储的思考
- 嵌入式Linux的发行版
- 小结

通常,理解一项给定任务的最佳途径是首先培养大局观。对于嵌入式系统开发的初学者来说,大量的基本概念是他们所面临的挑战。本章将带你快速浏览典型的嵌入式系统以及开发环境,重点放在概念以及组件上,它们使得系统开发工作独特而又富有挑战性。

2.1 需要嵌入式系统吗

嵌入式系统总是和几项关键的属性联系在一起,我们一般不会把PC叫作嵌入式系统。比方说,在远程数据处理中心有一个桌面PC硬件平台,用于监控某些危险的任务,并且需要根据任务状态决定是否报警。假设这类远程数据处理中心使用的都是无人监护的设备或者系统,那么这类系统就会有一些特殊的需求。例如,如果系统掉电之后又恢复供电,我们也希望相应的设备或者系统能够在无人干预的情况下自动恢复其工作。

嵌入式系统的外观或者尺寸多种多样,例如大型磁盘存储阵列或者电网供电系统,也有微小的模块,例如常见的MP3播放器或者手持移动终端设备等。这些嵌入式系统都多多少少具有以下特性:

- 具有处理引擎,例如通用微处理器;
- 其设计针对某类应用或者目的;
- 具有一些简单的(或复杂的)人机接口,例如汽车发动机的点火控制器;
- 一般是资源受限的,例如很小的内存占用(footprint)但没有硬盘驱动器;
- 可能对功耗有一定的限制,例如需要依靠电池供电的设备;
- 一般不采用通用计算平台;
- 软件通常内置,无需用户选择;
- 软硬件集成发布;

□ 通常独立工作，无需人工干预。

大多数情况下，相比普通桌面PC机，嵌入式系统都是资源受限的计算机系统。嵌入式系统只有有限的内存、很小的硬盘存储空间，甚至没有硬盘驱动器，有些时候也没有与外部交互的网络连接。很多时候，嵌入式系统唯一的人机交互接口就是串行通信端口或者几个简单的LED。这些以及其他问题都是嵌入式系统开发人员必须面对的。

BIOS 和引导装入程序

为桌面计算机加电后，一种叫作BIOS的软件系统会立即接管对系统处理器的控制。[BIOS是Basic Input/Output Software（基本输入输出软件）的缩写，但实际上，BIOS在计算机系统中发挥的作用已经远远超过了最初设计该软件时的目标，而且也越来越复杂。]通常，BIOS系统都是保存在闪存（Flash）里面的（稍后讨论），以便对BIOS软件进行必要的升级工作。

BIOS是一组复杂的系统配置软件例程，它记录了计算机硬件体系结构的底层细节信息。大多数用户并没有感觉到BIOS软件及其功能的存在，但是BIOS却是桌面系统不可或缺的一部分。在系统加电的时候，BIOS会首先接管对系统处理器的控制，它最主要的任务就是初始化系统硬件，特别是内存子系统，然后将操作系统从PC的硬盘驱动器中读出并加载。

典型的嵌入式系统中（这里假设嵌入式系统不是那类基于工业x86 PC系统硬件平台的计算机系统），有一种称为引导装入程序（bootloader）的软件程序来完成和BIOS一样的功能。在开发用户定制的嵌入式系统时，开发人员的一部分开发工作，就是要开发针对特定开发板的引导装入程序。幸运的是，现在有几种不错的开源引导装入程序可以供大家参考，开发人员可以自定义这些源代码，应用到自己的项目当中。有关引导装入程序的详细介绍参见第7章。

引导装入程序在系统加电之后需要完成下面几项比较重要的任务：

- 初始化关键硬件组件，例如SDRAM控制器、I/O控制器以及图形控制器等；
- 初始系统化内存，并且准备将系统控制权移交给相应的操作系统；
- 分配系统资源，例如内存以及外设控制器的中断电路等；
- 提供相应的机制，用于定位和加载操作系统映像；
- 加载操作系统，并将系统控制权交给操作系统，将必要的启动信息，例如系统全部内存块的数量、尺寸，串行通信端口的速度以及其他底层硬件配置数据等，传递给操作系统。

上述是对典型嵌入式系统的引导装入程序功能的简要概述。读者需要牢记的是：如果在用户自定义的平台上开发嵌入式系统，相应的引导装入程序必须由开发人员提供。如果在商业现货（COTS）平台（例如ATCA体系结构系统^①）上开发嵌入式系统，引导装入程序一般已经包含在了开发板当中，在第7章中将详细讨论有关引导装入程序的细节。

2.2 嵌入式系统剖析

图2-1展示了一套典型的嵌入式系统的框图，图中的系统是一个高级硬件体系结构的简单示

^① 有关ATCA平台将在第3章中详细介绍。

例，这种体系结构可以在无线终端接入点系统中见到。这个系统的核心是一个32位的RISC（精简指令集计算机）处理器，其中的闪存（Flash Memory）用于保存不再进行修改变化的应用程序以及数据。这里的主内存是SDRAM（同步动态随机存储器），根据系统应用程序的不同，其容量可能从几兆字节到上百兆字节。系统包含了一个实时时钟模块，这种模块通常由系统电池支持，用来保存时间信息，例如日历或者墙上时钟（包括日期）等。图中示例也包含了以太网和USB接口，同时还包含了串行通信端口，可以用作控制台通过RS-232进行通信。802.11芯片组实现了无线调制解调器的功能。

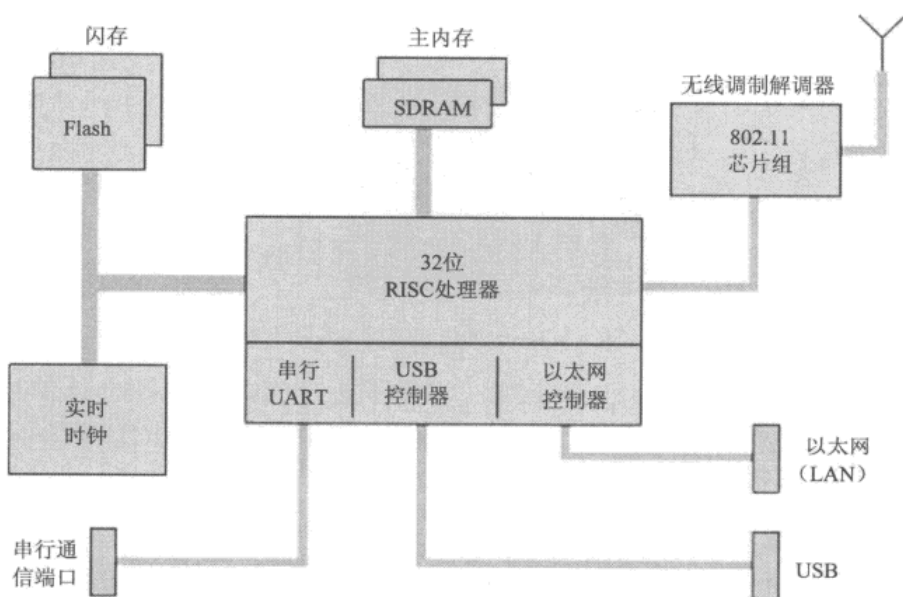


图2-1 嵌入式系统实例

与传统的CPU相比，嵌入式系统处理器承担的工作更多。例如，图2-1中所示的处理器，它包含了用作串行接口的集成UART接口、集成USB控制器和以太网控制器。很多处理器都会集成不同的外设，在第3章中将介绍几种集成了外设的处理器。

2.2.1 典型嵌入式 Linux 系统设置

几乎所有嵌入式Linux操作系统的初学者最先提出的问题，是开始开发工作都需要些什么。为了回答这个问题，首先要看一下典型的嵌入式Linux操作系统开发环境的设置，如图2-2所示。

图2-2展示了非常常见的一种系统设置。首先，系统包含了一个主机开发系统，它可以运行你喜欢的Linux发行版，例如Red Hat Linux、SuSE Linux或者Debian Linux。嵌入式Linux操作系统的目标板通过RS-232串行通信电缆与主机开发系统相连。图中将目标板的以太网接口与本地以太网集线器或者交换机连接起来，而同时主机开发系统也通过以太网与本地以太网集线器连接。在主机开发系统中需要包含开发工具和实用工具，以及由嵌入式Linux发行版提供的目标文件。

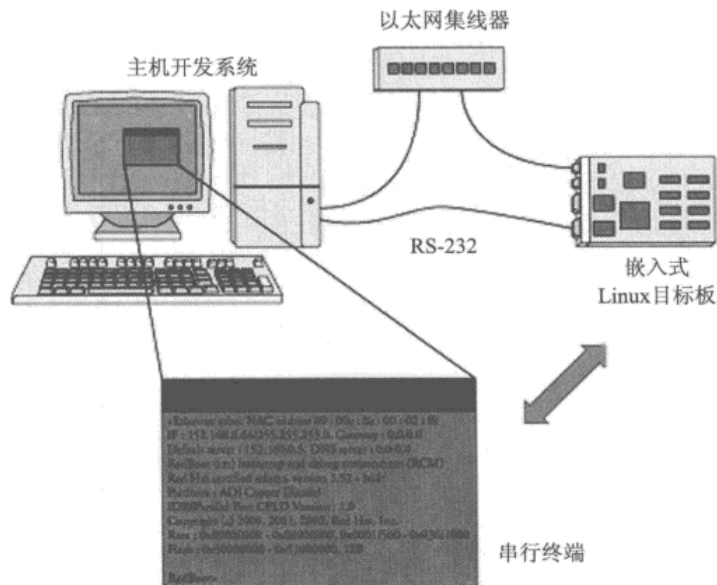


图2-2 嵌入式Linux操作系统的开发环境设置

在这个示例中，是通过RS-232与嵌入式Linux目标机进行连接的。串行终端程序用于实现与目标板的通信。Minicom是最常用的串行终端应用程序之一，几乎可以从所有桌面Linux发行版中获得。

2.2.2 启动目标板

当系统加电时，目标板所带的引导装入程序将立即接管系统处理器的控制权。它需要完成几个最基本的底层硬件初始化工作，包括处理器和内存空间设置，UART控制器（即串行通信端口）的初始化，以太网控制器初始化。代码清单2-1显示了目标板加电之后的启动过程中，从串行通信端口读回的一些信息。在本示例中，使用的目标板是AMCC公司的产品，一款具有PowerPC 440EP处理器的评估板，称作Yosemite。该产品基本上是AMCC 440EP嵌入式处理器的参考设计，AMCC公司为其预装了U-Boot 引导装入程序。

代码清单2-1 初始引导装入程序的串行端口输出信息

```
U-Boot 1.1.4 (Mar 18 2006 - 20:36:11)

AMCC PowerPC 440EP Rev. B
Board: Yosemite - AMCC PPC440EP Evaluation Board
VCO: 1066 MHz
CPU: 533 MHz
PLB: 133 MHz
OPB: 66 MHz
EPB: 66 MHz
PCI: 66 MHz
```

```

I2C:   ready
DRAM:  256 MB
FLASH: 64 MB
PCI:    Bus Dev VenId DevId Class Int
In:     serial
Out:    serial
Err:    serial
Net:    ppc_4xx_eth0, ppc_4xx_eth1

```

```
=>
```

当Yosemite板加电后，U-Boot执行一些底层硬件初始化工作，包括配置串行通信端口。然后，U-Boot输出了一条信息，该信息就是代码清单2-1的第1行。接下来显示系统的处理器名称和版本，然后是一些表征目标板特性的文本信息。这些信息都是U-Boot的开发者在源代码中加入的。

U-Boot还显示了一些内部时钟配置信息（内部时钟配置是在显示串行输出信息之前配置的）。当这些信息显示结束之后，U-Boot配置了各个硬件子系统，这里看到了I2C、DRAM、Flash、PCI以及网络子系统，这些子系统都是由U-Boot配置的。最后，U-Boot进入控制台模式，等待用户输入指令，此时可以看到=>命令提示符。

2.2.3 启动内核

现在，U-Boot完成了硬件、串行端口和以太网网络接口的初始化工作，剩下的唯一工作就是加载并启动Linux操作系统内核。所有的引导装入程序都提供了一条命令来加载并执行操作系统映像，代码清单2-2列出了U-Boot加载操作系统最常见的一种方法：手动加载并且启动Linux操作系统内核。

代码清单2-2 加载Linux操作系统内核

```

=> tftpboot 200000 uImage-440ep
ENET Speed is 100 Mbps - FULL duplex connection
Using ppc_4xx_eth0 device
TFTP from server 192.168.1.10; our IP address is 192.168.1.139
Filename 'uImage-amcc'.
Load address: 0x200000
Loading: #####
done
Bytes transferred = 962773 (eb0d5 hex)

=> bootm 200000
## Booting image at 00200000 ...
Image Name:   Linux-2.6.13
Image Type:   PowerPC Linux Kernel Image (gzip compressed)
Data Size:    962709 Bytes = 940.1 kB
Load Address: 00000000
Entry Point:  00000000
Verifying Checksum ... OK
Uncompressing Kernel Image ... OK
Linux version 2.6.13 (chris@junior) (gcc version 4.0.0 (DENX ELDK 4.0 4.0.0))

```

```

└─ #2 Thu Feb 16 19:30:13 EST 2006
AMCC PowerPC 440EP Yosemite Platform
...
< Lots of Linux kernel boot messages, removed for clarity >
...
amcc login:    <<< This is a Linux kernel console command prompt

```

tftpboot命令指示U-Boot利用TFTP^①协议通过网络将操作系统内核映像uImage-440ep加载到系统内存中。此时，操作系统内核映像是保存在主机开发系统的工作站中，也就是通过串行通信端口连接到目标板的计算机。同样，tftpboot命令传递的地址是目标板内存的物理地址，操作系统内核就被加载到首地址为这个地址的内存空间中。现在你不必详细了解这些细节，第7章中将继续介绍U-Boot。

下面将执行bootm（意为从内存映像启动）命令，指示U-Boot启动刚刚被加载到指定物理地址内的操作系统内核，这个地址也需要通过bootm命令行来指定。这条命令的执行将控制权从引导装入程序转移给Linux内核。假设Linux内核已经准确无误地加载并且配置，那么启动Linux内核的结果就是在控制台中显示一个命令提示符，即显示登录提示符。

需要注意的是，bootm命令结束了U-Boot引导装入程序的使命。这是一个非常重要的概念。与一般桌面PC中的BIOS不一样，大多数嵌入式系统都被配置成了这种方式，也就是当Linux内核接管系统之后，引导装入程序的使命即宣告结束，退出相应的引导装入程序应用程序。Linux操作系统内核将从引导装入程序手中接管所有的内存以及系统资源。将系统的控制权交还给引导装入程序的唯一方法就是重新启动嵌入式系统的开发板。

代码清单2-2中包含的下面这行串行端口输出信息是由U-Boot产生的：

```
Uncompressing Kernel Image ... OK
```

剩下的启动信息由Linux内核产生，本书将在后续章节中详细介绍这些信息。现在需要牢记的是：至此，引导装入程序已经结束了其系统启动的使命，余下的工作将由Linux内核完成。

2.2.4 内核初始化概述

当Linux内核开始执行后，它将向控制台输出大量状态信息，以使用户了解系统的启动过程。在这里讨论的例子中，Linux内核至少输出了超过100行的信息，直到显示系统登录提示符（代码清单2-2中省略了这些信息，以便更好地关注所讨论的话题）。代码清单2-3中显示了在显示系统登录提示符之前Linux内核显示的最后几行信息。注意，这里并不意味着要详细讨论内核初始化的细节（相应的细节讨论将在第5章进行），而是让你能够从比较高的层次了解在嵌入式系统加载Linux内核的过程中究竟发生了什么，以及启动Linux内核所需的组件。

代码清单2-3 Linux内核加载的最后几行信息

```

...
Looking up port of RPC 100003/2 on 192.168.0.9
Looking up port of RPC 100005/1 on 192.168.0.9

```

① 有关TFTP和其他将要使用的服务器软件将在第12章中详细介绍。

```
VFS: Mounted root (nfs filesystem).  
Freeing init memory: 232K  
INIT: version 2.78 booting  
...  
  
coyote login:
```

就在控制台终端显示登录提示符之前，Linux操作系统内核挂载了根文件系统（root file system）。在代码清单2-3中，Linux通过几个不同的步骤，将Linux的根文件系统通过以太网从远程NFS^①服务器中挂载进来，NFS服务器位于IP地址192.168.0.9所指定的计算机上。通常情况下，这台计算机就是系统的主机开发系统工作站。根文件系统通常包含了很多应用程序、系统库以及组成GNU/Linux系统的各种实用工具。

这里需要牢记的重点是，Linux需要一个文件系统。很多早期的嵌入式操作系统并不需要文件系统，这一点总是会让那些从事将早期的嵌入式操作系统移植到Linux嵌入式操作系统的工程师们感到惊奇与困惑。文件系统包含了预先定义的一组系统目录树以及文件，它们都保存到硬盘驱动器或者其他媒介之中，Linux内核将其挂载为根文件系统。

注意，Linux还可以从一些其他驱动器中挂载根文件系统，最常见的方式就是直接在系统的硬盘空间中挂载根文件系统，就像常见的桌面Linux工作站那样。事实上，利用NFS服务加载Linux操作系统内核是非常不错的选择，除非开发的嵌入式Linux产品无法与开发环境联系起来。在本书的后述内容中，大家将充分感受到从主机开发环境的NFS服务挂载内核的灵活与强大。

2.2.5 第一个用户空间进程：init

在继续讨论前，还有重要的一点需要说明。注意代码清单2-3中的下面这行信息：

```
INIT: version 2.78 booting.
```

直到目前，内核本身还在执行代码，在内核上下文（kernel context）中完成若干初始化工作。在这种操作状态下，内核拥有所有的系统内存，并且拥有使用所有系统资源的权限，内核能够访问所有物理内存以及所有的I/O子系统。

当Linux内核完成所有内部初始化工作并且挂载了根文件系统之后，默认将启动名为init的应用程序。启动init应用程序，就意味着系统将运行在用户空间（user space）或者用户空间上下文中。在这种操作模式下，用户空间进程就不像内核进程那样具有访问所有资源的权限，用户空间只具有有限的权限，必须使用内核系统调用来请求相应的内核服务，例如设备和文件I/O。所有用户空间进程、应用程序都在虚拟内存空间中执行，该虚拟内存空间由内核随机分配^②和管理。而内核需要与处理器中专用的内存管理单元配合，完成用户空间进程虚拟内存地址到物理内存地址的转换工作。这种体系结构的一个最大好处就是，用户进程的内存空间错误不会引起其他进程的内存空间错误。但是在遗留嵌入式操作系统中，这是常见的隐患，可能导致难以检测的错误。

^① NFS和其他需要的服务器将在第12章中讲述。

^② 不是真实意义上的随机，这里提到随机只是介绍的需要。这个问题在后面还有详细讨论。

你不必因为对这些概念不熟悉而感到紧张，本节的目的是给出总览，在进一步的阅读中，你将会获得更详细的知识。

2.3 存储的思考

嵌入式系统的一个最具有挑战性的方面，就是可利用的硬件资源十分有限。虽然在奔腾4计算机中拥有180 GB的硬盘空间是很平常的事情，但你很难找到拥有如此巨大硬盘容量的嵌入式系统。通常我们会用成本低、容量小的稳定存储设备来代替硬盘。硬盘体积过大、有旋转部件、对震动敏感、需要多电源供电，这些缺点使得硬盘不适合在嵌入式系统应用。

2.3.1 闪存

几乎所有人都熟悉CompactFlash[®]模块，它广泛应用在数码相机、PDA（两者都是非常典型的嵌入式系统的例子）等个人消费类设备中。这些模块在功能上可以看做固态硬盘驱动器，可以在很小的区域中存储几百兆甚至几千兆的数据。它的内部没有活动部件，可以在相对颠簸的环境中工作，而且只用一个普通电源就可以驱动。

目前，有几家从事闪存制造的公司，他们制造了不同封装和不同容量的闪存。在嵌入式设备中只安装1 MB或2 MB非易失存储器的情况比较普遍，而更为典型的存储需求是：在嵌入式设备上安装4 MB~256 MB，甚至更大容量的存储器。目前越来越多的嵌入式系统正在安装1 GB以上的非易失存储器。

闪存可以在软件的控制下执行写操作和擦除。虽然硬盘仍然是写速度最快的存储设备，但是经过了一段时间的改进，闪存的写操作及擦除的速度都大大提高了，当然相对于硬盘来说，速度还是很慢。为了正确使用它们，我们有必要了解一下硬盘与闪存的本质区别。

闪存可以被划分为一个个较大的可擦除单元，称为擦除块（erase block）。它的一个显著特点就是对闪存数据进行写入及擦除的方式。在普通的闪存芯片中，可以用软件改变数据，从二进制的1改成二进制的0，一次改变一位，但是如果想把一位0改回1，那么就要对整个块进行擦除操作。正因如此，闪存中的块一般都被称为擦除块。

典型的闪存都包含若干个擦除块。例如，一个容量为4 MB的闪存芯片要包含64个大小为64 KB的擦除块。有些闪存还可以拥有大小不等的擦除块，以便于数据存储的灵活布局。这些闪存往往被称为支持启动扇区或启动块的闪存。通常情况下，引导装入程序被保存在较小的块中，而内核和其他一些数据则保存在较大的块中。图2-3展示了一个低地址启动闪存中各块大小的布局图。

为了修改存储在闪存中的数据，被修改数据所在的块要被全部擦除。即使仅仅修改某个块中的一个字节，整个块也必须被擦除并重写^②。相对于硬盘的扇区来说，闪存块的尺寸比较大。高性能硬盘通常具有512 B或1024 B的写扇区。由此可见，更新闪存中数据的次数往往是更新硬盘中数据的次数的好几倍，这是因为每次对闪存中数据更新时，必须写回大量的数据。在最坏情况下，这种写操作的周期可能持续好几秒钟。

① 详细介绍请参考www.compactflash.org。

② 记住，你可以一次把一个字节的1改成0，但是，你要把任意一位从0写回1，必须擦除整个块。

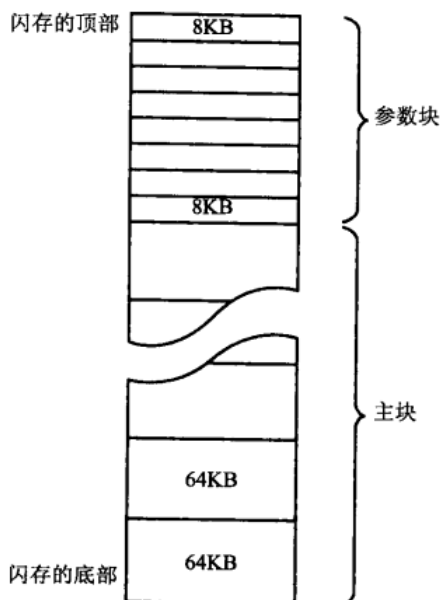


图2-3 具有启动块的闪存的体系结构

闪存另一个必须考虑的局限性是闪存单元的可写次数。闪存单元的可写次数是有限的。虽然可写的次数相当多（通常每个块可以写100K次左右），但是不难想象，不良的存储方案（即使是个bug）可以很快毁掉闪存。由此可见，在进行设计时，一定要避免把系统日志的输出定位在基于闪存的存储设备中。

2.3.2 NAND 闪存

NAND闪存采用了一种相对新的闪存技术。在NAND闪存投放市场时，前面介绍的传统闪存就被称为NOR闪存。它们的不同点在于其内部内存单元的体系结构。NAND闪存通过提供小尺寸的块改进了传统（NOR）闪存上的一些限制，它可以更快更高效地进行写操作，同时大大提高了闪存阵列的使用效率。

NOR闪存为微处理器提供的接口方式与许多微处理器外设提供的方式极为相似。那就是它们具有并行的数据和地址总线，可以与微处理器的数据/地址总线直接连接^①。闪存阵列中的每个字节或字都可以通过地址随机访问。相反，NAND闪存中的数据必须按照各厂商定义的复杂接口进行串行访问。NAND闪存提供了一个与传统硬盘及相关控制器极为相似的操作模型。所有的数据访问都是以串行突发的形式进行的，数据规模要比NOR闪存中的块小得多。虽然NAND闪存的可擦写次数比NOR闪存要少很多，但是其写操作的寿命要比NOR闪存的高出几个数量级。

综上所述，NOR闪存可以被微处理器直接访问，代码可以在NOR闪存中直接运行。（基于性能考虑，这种做法很少见，但是在资源十分有限的系统中，往往采用这种做法。）实际上，很

① 直接连接是逻辑意义上的。实际电路中可以通过总线驱动器或桥设备等进行连接。

多处理器不能像在DRAM中那样缓存访问过闪存的指令，这就会大大降低运行速度。相比之下，NAND闪存更适合用于文件系统的存储之用，而不是存储原始二进制的可执行代码和数据。

2.3.3 闪存的用途

嵌入式系统的设计者在考虑闪存的布局及用法时可以有多种选择。在最简单的系统中，资源是不做过多限制的，原始二进制数据（也许是压缩过的）可以存储在闪存上。当启动时，存储在闪存中的文件系统的映像被读到Linux ramdisk块设备中，接着挂载成文件系统且只能从RAM中访问。当存储在闪存中的数据不需要更新，或者需要更新的数据相对于ramdisk的尺寸相当小时，以上设计就是一个很好的选择。需要特别注意的是，在ramdisk中被更改的文件在系统重新启动或重新加电之后是不会被保存的。

图2-4展示了一个简单的嵌入式系统中，在需要保存的动态数据很少且更新不频繁的情况下的闪存的组织架构图。

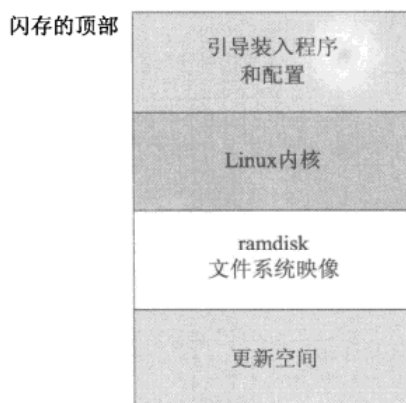


图2-4 闪存结构图

引导装入程序通常安排在闪存阵列的最顶端或最底端。紧随引导装入程序的是Linux内核映像，以及ramdisk文件系统映像^①，这个文件系统就是根文件系统。通常情况下，Linux内核映像与ramdisk文件系统映像都是压缩过的，引导装入程序在启动过程中负责对它们进行解压。

为了存储在重启或重新加电情况下需要保存的动态数据，需要专门分配一小块闪存空间，或者提供另外一种非易失存储介质^②。对于需要保存配置信息的嵌入式系统来说，这是很典型的配置形式。例如在消费市场上常见的无线AP产品，就采用了这种形式。

2.3.4 闪存文件系统

前面所描述的简单闪存布局将不可避免地带来一些不利限制，而这些可以通过在闪存中引入

^① ramdisk文件系统将在第9章中详细介绍。

^② 实时时钟模块通常包含少量非易失数据存储。使用串行EEPROM是替代少量非易失数据存储的一种常用选择。

与硬盘相似的文件系统管理数据的方案来解决。在闪存设备上实现的早期文件系统包含了一个能够模拟普通硬盘的512 B扇区的简易块设备层。这些简易的模拟层允许数据以文件的形式而不是无格式的大数据块的形式进行访问，这同时也带来了访问性能的限制。

对闪存文件系统的一个首要改进是引入了均衡读写机制（wear leveling）。就像前面所讨论的，闪存块具有有限的擦写次数。均衡读写算法可以把写操作均匀地分布到闪存的各个物理擦除块中。

闪存文件系统所带来的另一个缺陷，是在电源故障或非正常关机的情况下丢失数据的风险。考虑闪存块的尺寸相对很大，而被写入到闪存块中的平均文件尺寸又相当小时的情况。从前面已经知道，闪存块的写操作一次必须以一个块为单位来完成。因此，要想写入一个8 KB的小文件，必须擦除并重写整整一个64 KB或128 KB大小的闪存块，在最坏的情况下，这个操作可能需要几十秒钟来完成，这么长的时间就为电源故障引起数据丢失埋下了极大的隐患。

目前较为常用的闪存文件系统之一就是JFFS2，即Journaling Flash File System 2。它具有几个重要的特性，可以提高整体运行性能，延长闪存使用寿命，降低由电源故障引起的数据丢失风险。在最新版本的JFFS2文件系统中，较为显著的改进有：更高一级的均衡读写机制；数据的压缩及解压，此功能可以保证在一定大小的闪存中存储更多的数据；支持Linux下的硬链接。我们将在第9章和第10章中对此做更详细的介绍。

2.3.5 存储器空间

实际上所有的遗留嵌入式操作系统都把系统存储器看作是一个大的线性地址空间来管理。这也就是说，一个微处理器的地址空间应该从0开始直到它的物理最大地址。例如，如果一个微处理器有24条物理地址线，它的最高存储器地址将是16 MB，因此它的16进制地址范围应该从0x00000000到0x00ffffff。在设计硬件时，通常把DRAM安排在地址空间的底部，而闪存则安排在顶部。DRAM之上与闪存之下的未用空间通常安排给板上集成的外设芯片使用。这种设计结构通常由微处理器文档来说明。图2-5是一个简单嵌入式系统的存储器地址空间设计实例。

在以遗留操作系统为基础的传统嵌入式系统中，操作系统与所有的任务^①拥有对系统资源同等的访问权限。一个进程中的错误可以很轻易地冲掉系统中其他地址空间的内容，被冲掉的地址空间可以是它自己本身的地址空间，也可以是操作系统或其他任务的地址空间，甚至可以是地址空间中的硬件寄存器地址。虽然以上设计具有简单易行的优点，但同时也给系统留下了出错的隐患，并且一旦出错将很难调试。

高性能的微处理器都拥有被称为内存管理单元（MMU）的硬件模块，它的目的是允许操作系统对它本身的地址空间及被分配给其他进程的地址空间实施高效的管理和控制。这种控制主要表现为两种形式：访问权限的控制和地址转换。访问权限允许操作系统限制某段地址空间只能被某些特定的任务访问，地址转换允许操作系统对它的地址空间实施虚拟化，虚拟内存的支持具有很多好处。

^① 在本讨论中，任务这个词指的是所有执行中的线程，不论它是如何产生、管理和调度的。

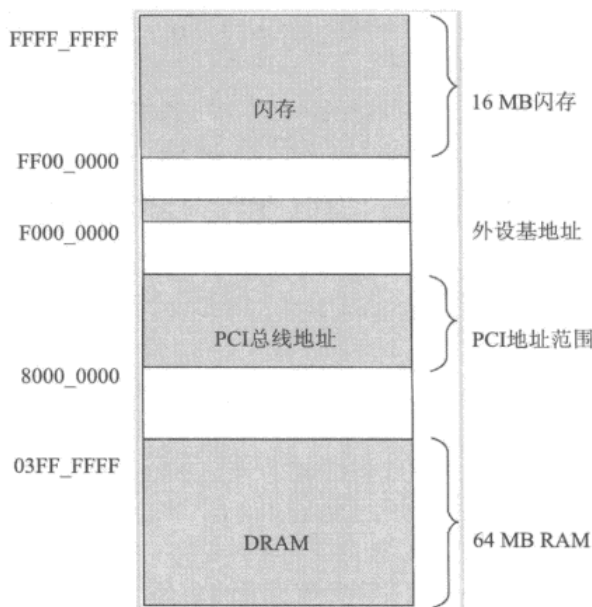


图2-5 典型的嵌入式系统地址空间布局图

Linux内核依靠硬件内存管理单元的优势实现了支持虚拟内存的操作系统。虚拟内存技术能够带来的最大好处是，可以更加有效地利用物理内存，并给用户远远大于实际物理内存的更大的可用地址空间。另一个好处是，内核可以为分配给某个任务或进程的地址空间设置访问权限，以阻止一个进程由于误操作而非法访问其他进程或整个操作系统的地址和资源。

让我们来看看它是如何工作的。对虚拟内存系统整体性的介绍已经超出了本书的范围^①，在这里我们将按照嵌入式系统开发者在实际工作中所接触的顺序，来逐一介绍虚拟内存的相关知识。

2.3.6 运行上下文

在Linux启动运行的最初阶段，必须要做的一项工作，就是要配置好处理器的内存管理单元并初始化与之配套的数据结构，以支持虚拟地址到物理地址的转换。当这一步完成之后，内核就运行在它自己的虚拟地址空间中了。在最新的版本中，内核开发人员规定的内核虚拟地址默认为0xC0000000。在大多数体系结构中，这个地址被设置成可配置参数^②。如果我们看一下内核符号表，将会发现所有的内核符号都以0xC0xxxxxx来编址。由此可见，当内核在内核空间执行代码时，处理器的IP指针都将指向这个地址范围中。

在Linux中，根据指定线程^③的运行环境，我们可以把它分为两个独立的运行上下文。当线程完全运行在内核空间时，我们称之为内核上下文，而应用程序则运行在用户空间上下文。一个用

① 有许多优秀的参考书都对虚拟内存系统作了详细介绍，见本章“参考资源”所列举的参考书。

② 但是一般情况下不做改动。

③ 线程这个词在这里指通常意义上的任何一段指令流。

用户空间的进程仅能访问属于它的地址空间，而要访问文件或设备I/O等特权资源则要通过系统调用。用实例可以使上面的介绍更加清楚。

假设一个应用程序打开了一个文件并且发出了读取请求（如图2-6所示）。读文件的函数开始于C库中的`read()`函数，它处于用户空间，接着它将向内核发送读请求。这个读文件的操作将引起用户空间上下文与内核空间上下文的切换，以保证完成文件数据的读请求。在内核内部，这个读文件的请求最终将产生对数据所在磁盘扇区的物理访问。

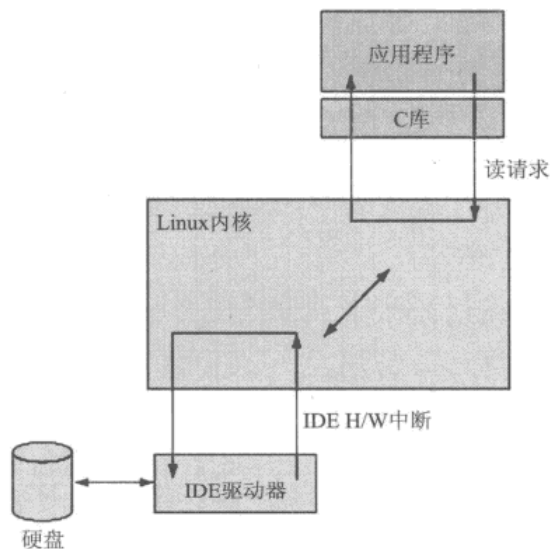


图2-6 文件读操作的调用结构图

通常情况下，硬盘的读操作被异步施加给硬件本身。异步的意思就是，读操作的请求会首先通知到硬件，之后处理器去做其他工作，当请求的数据准备好以后，再通过中断的形式通知处理器来读，请求此数据的应用程序将一直被阻塞在等待队列中，直到数据有效时为止。当硬盘的数据有效后，它将向处理器发送一个中断请求（这样简化的描述就是为了便于读者理解），当内核收到硬件中断请求后，它将暂时挂起当前正在执行的进程以读取硬盘中的数据。以上是一个运行在内核上下文的线程实例。

归纳来讲，我们已经认识了两个运行上下文，用户空间上下文与内核上下文。当应用程序执行系统调用导致上下文的切换并进入内核后，对此进程来说它就是在执行内核的代码，这个就被称为内核的进程上下文。而处理硬盘操作的中断处理例程（ISR）是一段不被任何进程直接调用的内核代码。在这段特殊的上下文中有的一些限制条件，包括不能阻塞（休眠）或调用可能导致阻塞的内核功能。想要进一步理解这个概念，请参考本章的“参考资源”。

2.3.7 进程中的虚拟内存

当用户在Linux命令提示符下输入`ls`，就将产生一个进程，内核将为这个进程分配存储器资

源，并且会分配一段虚拟地址空间。被分配的地址空间与内核中的地址没有固定关系，也与任何其他运行的进程无关。而且，进程所用的虚拟地址空间与目标板上的物理地址空间也没有直接的关系。事实上，由于分页和缓存交换的机制，一个进程在它的生命期内将占据多块不同的物理地址空间。

代码清单2-4提供了一个简单的“Hello World”型的程序段来阐述前面介绍的概念。在这个例子中将会展示内核是如何为这个进程分配地址空间的。这段代码编译后运行在前面提到的AMCC Yosemite目标板上，这块板上拥有256M DRAM存储器。

代码清单2-4 嵌入式系统风格的Hello World

```
#include <stdio.h>

int bss_var;          /* Uninitialized global variable */

int data_var = 1;     /* Initialized global variable */

int main(int argc, char **argv)
{
    void *stack_var;   /* Local variable on the stack */

    stack_var = (void *)main; /* Don't let the compiler */
                                /* optimize it out */

    printf("Hello, World! Main is executing at %p\n", stack_var);
    printf("This address (%p) is in our stack frame\n", &stack_var);

    /* bss section contains uninitialized data */
    printf("This address (%p) is in our bss section\n", &bss_var);

    /* data section contains initialized data */
    printf("This address (%p) is in our data section\n", &data_var);

    return 0;
}
```

代码清单2-5展示了运行上面的程序后在控制台所产生的输出。请注意，hello进程在256MB边界之上的高端内存中运行（0x10000418），而且栈地址几乎已经到了32位地址空间的中部（0x7ff8ebb0）。怎么会这样呢？DRAM在系统中都是连续编址的，在普通观察者看来，我们好像有2GB大小的DRAM可用。实际上这些地址空间是由内核分配，并由Yomesite目标板上的256MB物理RAM在背后作映射支持的。

代码清单2-5 hello程序的输出

```
root@amcc:~# ./hello
Hello, World! Main is executing at 0x10000418
This address (0x7ff8ebb0) is in our stack frame
This address (0x10010a1c) is in our bss section
This address (0x10010a18) is in our data section
root@amcc:~#
```

在支持虚拟内存的系统中，一个显著的特点就是当可用物理RAM低于某一个设计阈值时，

内核会把内存页置换到大的存储介质中，通常是硬盘。内核会定期检查内存区域的活跃程度，找出哪些内存页在最近被用到的次数最少，以便把它置换到硬盘中，然后释放当前进程所占的空间。嵌入式系统的开发者出于对性能及资源的考虑，通常都会在嵌入式系统中禁止置换的功能。例如，如果使用访问很慢并且可擦写寿命有限的闪存作为置换设备，那将是多么糟糕的设计。没有了置换设备，在设计应用程序时要格外小心，要保证自己的应用存在于有限的可用物理存储器中。

2.3.8 交叉开发环境

在为嵌入式系统开发应用程序或设备驱动程序之前，需要一套工具（编译器、二进制工具等）来为我们的目标系统生成可执行的二进制代码。考虑一下PC机上的简单应用程序，例如比较典型的“Hello World”例子。当在PC机上编完源码之后，你将调用桌面系统中嵌入的（单独购买并安装的）编译器来生成可运行的二进制代码。这套二进制代码最终将运行在编译它的主机上，这称为本机编译（native compilation），它的意思就是使用本机的编译器进行编译，产生出的代码也将运行于本机之上。

注意一点，本机的意思不是特指某一种体系结构，事实上，如果你拥有一套可以在目标板上运行的二进制工具链（tool chain），完全可以为你的目标体系结构做本地化应用程序编译。其实测试一个新内核版本或新目标板的方法就是反复地在其上进行编译。

在交叉开发环境中开发软件，其实就是要让本机上运行的编译器产生出不兼容于本机的可执行二进制格式。交叉开发工具存在的主要原因是资源的限制（内存大小，CPU性能），在嵌入式系统中做本地化的软件开发和编译是不现实的，也是不可能的。

这其中有无数的陷阱阻碍了很多新手迈向嵌入式开发领域。当编译一个程序时，编译器通常都知道如何找到所需的头文件以及正确编译所需的链接库。为了说明这些概念，再来看看“Hello World”这个程序，代码清单2-4中所列的例子将以下面的命令进行编译：

```
gcc -Wall -o hello hello.c
```

在代码清单2-4中，我们看到一个头文件stdio.h，这个文件没有与gcc命令行指定的hello.c文件处在同一个目录下，那么编译器是如何找到它们的呢？另外printf()这个函数没有定义在hello.c的文件中，因此当hello.c编译时，将会产生一条“无法引用的符号链接”的报错，链接器在链接时是如何解决这个问题的呢？

编译器内部有默认设置来规定头文件的搜寻目录，当遇到包含头文件的语句时，编译器会按照它的默认地址链表来搜索这个文件，链接器搜索待链接符号printf()的过程与此类似。链接器在默认情况下到C库（Libc-*）中寻找待链接符号，这条规则也是被定制在二进制工具链中作为默认设置的。

现在考虑一下，假如你要为基于PowerPC体系结构的嵌入式系统开发一个应用程序。首先你需要有一个交叉编译器来生成PowerPC体系结构的二进制代码。例如编译上面的hello.c的例子，你需要为交叉编译器输入与前面相似的编译命令。编译之后，很有可能由于链接器在寻找printf()这个待链接符号时而错误地引用了x86版本的C库。当然，运行这个既包含了x86又包含

了PowerPC二进制指令的混合程序^①的后果是很明显的，那就是程序崩溃。

解决这个问题的方法，就是指导交叉编译器到非默认路径中寻找头文件和所需的链接库。我们将在第12章中对此做更详细的阐述。在这里举这个例子的目的，就是为了勾画出本地开发环境与针对嵌入式系统的交叉开发环境的区别。在第14章中可以看到同样的问题与解决方法也适用于交叉调试的问题。在第12章中将会看到一个正确的交叉开发环境是成功开发的关键，当然正确的交叉开发环境不仅仅只包含编译器。

2.4 嵌入式 Linux 的发行版

确切地说，什么是发行版呢？Linux内核启动之后，它期望找到并挂载一个根文件系统。当一个适当的根文件系统被成功挂载之后，启动脚本将启动一批系统需要的程序及工具软件。这些程序通常都要调用其他一些程序来完成特殊任务，例如产生登录界面、初始化网络接口、启动用户程序等。上述每一个程序都提出了特定的系统需求。大多数Linux应用程序依靠一个或多个系统库，其他一些程序还需要配置或日志文件的支持。总之，即使一个很小的嵌入式Linux系统，也需要准备一个目录结构完整且包含了所有相关文件的根文件系统。

一个功能强大的桌面系统在它的根文件系统中通常拥有成千上万个文件。这些文件分别来自按功能组织的各个安装包中，而这些安装包通过一个包管理器来管理和安装。Red Hat公司的安装包管理工具（rpm）是一个流行的工具，它广泛应用在Linux系统组件的安装、卸载与更新中。如果你的Linux工作站是基于Red Hat产品的，例如Fedora Core系列，就可以输入rpm-qa命令来列出系统中已经安装的所有软件包。

一个安装包可以包含许多文件，实际上有些安装包中已经包含了好几千个文件。一个完整的Linux发行版能够包含数百个或上千个这样的安装包，下面列举一些在嵌入式Linux发行版中常见的安装包及其功能说明。

- `initscripts`——包含了基本的系统启动及关机脚本；
- `apache`——实现了广受欢迎的Apache Web服务器；
- `telnet-server`——包含了实现telnet服务器所需的文件，它可以为嵌入式系统建立远程登录会话；
- `glibc`——标准的C库；
- `busyBox`——Linux/Unix系统中常用的命令行工具的精简集合^②。

这就是一个Linux发行版所表示的涵义。一个典型的Linux发行版通常包含了好几张装满了各种应用程序、函数库、工具集以及文档的CD光盘。在安装一个发行版Linux的时候，用户可以按照默认设置来安装一个包含完整功能的系统，同时它也可以被裁减安装以满足特殊的功能需求。也许你已经熟悉了一款常见的桌面Linux发行版，例如RedHat或Suse。

嵌入式系统的Linux发行版具有一些显著的不同点。首先，嵌入式发行版中的可执行代码不

^① 事实上这种程序很少被编译或链接，更谈不上运行。

^② 这个安装包十分重要以至于本书为它单独设立了一章，第11章将对它作更为详尽的阐述。

能在PC上运行，而只能在目标体系结构的处理器中运行（当然，如果你的嵌入式Linux的目标体系结构是x86，上面的限制将不适用）。一个桌面Linux发行版集成了许多GUI工具以满足大多数桌面系统的用户，例如图形版的时钟程序、计算器、个人时间管理工具、电子邮件客户端以及其他一些有用的工具。嵌入式Linux发行版则通常省略了这些组件，而集成了一些开发者所需的特殊工具，例如存储器分析工具、远程调试工具以及更多的类似工具。

其次，嵌入式发行版中通常包含的是交叉开发工具，而不是本机开发工具。例如，集成在嵌入式Linux发行版中的gcc工具链虽然运行在x86的PC机中，但是能够编译出在目标板运行的二进制代码。工具链中的其他工具与此相似，它们都运行在开发主机上（通常是x86的PC机），但是都是为其他系统体系结构（如ARM或PowerPC）服务的。

2.4.1 Linux 商业发行版

目前有几家商用嵌入式Linux的开发商，其中的领先者已经在嵌入式Linux领域纵横了许多年。Linuxdevices.com是一个深受大众欢迎的嵌入式Linux新闻及信息发布门户网站，已经为目前可用的商业版嵌入式Linux编辑了一份详尽的列表，虽然在某种程度上它遭到了一些非议，但对于初学者来说它还是一个不错的网站。你可以在www.linuxdevices.com/articles/AT9952405558.html网页浏览到它所编辑的列表。

2.4.2 Linux 自定义发行版

你可以根据自己的需要，为嵌入式系统选择Linux组件；同时你将不得不决定为此付出的冒险是否值得。如果你是兴趣使然的话，这种尝试对你来说是件好事。但是要知道你将会为此付出大量的时间才可以集成项目所需的工具及软件集，同时你要保证它们相互之间可以正常搭配工作。

对于初学者来说，你需要一套工具链。gcc和binutils可以从www.fsf.org网站或其他映像站点下载。这两个工具对于编译内核和应用程序来说是必需的。它们主要以源码的形式进行发布，在使用前必须针对目标系统架构进行配置和编译。针对这些工具的最新源码版本通常需要一些补丁文件，这种情况尤其适用于x86/IA32以外的体系结构。补丁文件通常可以在相应源码包所在的网站找到，你所面临的主要问题是，决定针对你的问题或体系结构应该选择哪个补丁文件。

2.5 小结

本章对许多问题进行了粗略的描述，学完本章之后你应该对后续章节内容有了正确的初步认识。在后续章节中，这种认识将协助你扩充知识和技能，以保证你在今后的嵌入式项目中取得成功。

- 嵌入式系统拥有一些共性。系统资源有限，用户接口简单或干脆没有，通常是为特定应用而设计。
- 引导装入程序是嵌入式系统中的一项关键组件，如果你的嵌入式系统基于一个用户定制设计的目标板，那么必须提供一个引导装入程序，通常这是对现有引导装入程序的移植。

- 为了启动一个定制的目标板，需要提供几个软件组件，包括引导装入程序和内核与文件系统的映像。
- 闪存存在嵌入式系统中被当作存储介质而广泛应用，我们将在第9章和第10章介绍闪存的知识。
- 应用程序（也称为进程）存在于内核给它分配的虚拟地址空间中。应用程序被称为运行在用户空间的程序。
- 一个配置正确的交叉开发环境对于嵌入式开发者来说极为关键，我们将在第12章对它做详细的阐述。
- 你需要一个嵌入式Linux发行版来开始你的嵌入式开发，嵌入式发行版中包含了许多针对你所选择的体系结构编译或优化过的组件包。

参考资源

Linux Kernel Development, 2nd Edition
Robert Love
Novell Press, 2005

Understanding the Linux Kernel
Daniel P. Bovet & Marco Cesati
O'Reilly & Associates, Inc., 2002

Understanding the Linux Virtual Memory Manager
Bruce Perens
Prentice Hall, 2004



本章内容

- 单机处理器
- 集成化处理器：片上系统
- 硬件平台
- 小结

本章将介绍一些基础知识，这些知识将使你能够在嵌入式处理器的浩瀚海洋中自由航行。我们也将介绍一些在市场上常见的处理器及其基本特性。首先重点介绍单机处理器，这些处理器是功能最强大的处理器，它们需要额外的芯片组来构成完整的嵌入式系统；然后介绍一些集成化处理器，集成化处理器类型繁多，这里介绍那些能够在嵌入式Linux操作系统下应用的处理器；最后介绍一些现在较为常见的硬件平台。

仅从表面上看，可供嵌入式系统设计开发选择的嵌入式处理器类型多达数十种。本章要介绍那些具有内存管理单元（MMU）硬件，并且支持Linux操作系统的处理器产品。Linux操作系统的一个基本体系结构，就是它是具有虚拟内存管理操作的操作系统^①，在不具备内存管理单元的处理器上应用Linux操作系统将不得不舍弃一些有用的操作系统内核的体系结构特性，这超出了本书的讨论范围。

3.1 单机处理器

所谓单机处理器是指那些专门用于处理功能的处理器芯片，相对于后面介绍的集成化处理器，单机处理器需要一些外围电路支持才能够实现其基本功能。大多数情况下，这就意味着需要围绕在处理器周边的芯片组或者自定义逻辑电路，来完成诸如DRAM控制、系统总线寻址配置、外围设备（例如键盘控制器、串行端口）控制等功能。单机处理器通常可以提供最高的综合CPU性能。

目前的处理器有32位处理器和64位处理器^②，这两类处理器在嵌入式系统中广泛使用，例如

^① Linux对那些没有MMU的处理器也提供了支持，但这并不是Linux的主流。

^② 这里的32位或者64位是指处理器内部主要设备的数据宽度，例如指令执行单元、寄存器、地址总线等。

IBM PowerPC 970FX、Intel Pentium M、Freescale MPC74xx主机处理器等。

本节将介绍几种由主流制造商生产的单机处理器，这些处理器能够非常好地支持Linux操作系统，并已经在很多嵌入式系统设计中采用。

3.1.1 IBM 970FX

IBM 970FX处理器核是一个具有64位处理能力的高性能单机处理器芯片。它采用超标量体系结构，也就是说该处理器核能够同时处理多条指令，完成指令的取指、发生以及获取结果。这是通过流水线（pipelining）体系结构实现的，它能够同时提供多条指令流，用于完成高效率的处理计算。IBM 970FX处理器最多可以包含25级流水线，这取决于各类指令流以及其中包含的操作。

下面列出了IBM 970FX处理器的一些关键特性：

- 基于流行的PowerPC体系结构的64位处理器；
- 深度流水线设计，针对极高计算性能的应用；
- 静态和动态电源管理特性；
- 多重休眠模式，以获得最小化供电需求和最大化电池寿命；
- 动态可调节时钟主频，支持低功耗模式；
- 针对高性能、低延迟存储管理进行优化。

IBM 970FX处理器已经被用于高端刀片服务器以及高性能计算平台中，包括IBM公司自己推出的刀片服务器平台。

3.1.2 Intel Pentium M

作为目前最流行的计算机体系结构，x86体系结构具有32位处理器和64位处理器（通常称作IA32和IA64），在大量嵌入式系统应用中都能找到这两种处理器。大多数情况下，这些系统解决方案的硬件平台基于不同的COTS（commercial off-the-shelf，商业现货）硬件实现，众多产品制造商能够提供x86单板计算机，以及不同形式的完整平台。3.2节将讨论一些现在广泛使用的平台。

Intel Pentium M处理器最为活跃的市场是当前的笔记本电脑市场，并已经在嵌入式产品中占有一席之地。与IBM 970FX处理器类似，Pentium M处理器也是超标量体系结构，这些特性对于嵌入式系统应用同样具有吸引力：

- Pentium M处理器基于流行的x86体系结构，因此得到大量软硬件厂商的支持；
- 与普通的x86处理器相比，Pentium M处理器功耗较低；
- 先进的电源管理特性能够支持低功耗工作模式，支持多重休眠模式；
- 动态调节时钟速度提高了电池供电的操作能力，例如待机；
- 根据芯片上的温度监控系统自动调节处理器进入低功耗模式，以降低功耗，防止过热；
- 多种主频和电压应用模式（可以动态选择）能够使便携设备的电池寿命最大化。

这些特性对于嵌入式系统开发非常有用，嵌入式系统常常考虑移动性需求或者电源功耗需求。正是由于Pentium M处理器具有良好的电源管理和温度管理特性，所以该处理器才会在这种应用中如此热门。

3.1.3 Freescale MPC7448

Freescale MPC7448 处理器通常被看作是第四代PowerPC处理器核，一般称为G4处理器^①。它是一种高性能32位处理器，在网络设备或者电信应用中比较常见。有几家制造商设计生产了符合工业标准平台规范的刀片产品，这些产品有些采用了此类处理器，有些则采用了类似的单机Freescale处理器。3.3节将介绍这几种常见的硬件平台。

MPC7448处理器在信号处理和连网应用等领域非常流行，因为它具有如下先进特性：

- 操作时钟速率可达到1.5 GHz；
- 具有1 MB的二级高速缓存；
- 具有高性能电源管理特性，支持多重休眠技术；
- 具有高级AltiVec向量执行单元；
- 可调节电压，降低功耗需求。

MPC7448处理器包含了Freescale公司的一项专利技术——AltiVec，能够实现快速算术计算以及其他数据处理计算应用。AltiVec单元包含了32个128位文件寄存器，AltiVec文件寄存器中的每个值可以看作是一个具有多个元素的向量。AltiVec寄存器定义了一组指令用于有效操纵这个向量数据，同时进行核心CPU指令操作。AltiVec操作包括求和—交叉计算、乘—求和计算、同步数据分布（存储）操作、数据收集（加载）操作等。

程序员可以利用AltiVec硬件加快信号处理或者网络设备中常出现的软件计算速度，例如实现快速傅里叶变换（FFT）、数字信号处理（如滤波）、MPEG视频编解码、快速生成加密协议（DES、MD5和SHA1）等。

Freescale公司还开发了一系列单机处理器，例如MPC7410、MPC7445、MPC7447、MPC745x以及MPC7xx系列产品等。

3.1.4 配套芯片组

前面介绍的几种单机处理器在真正的使用中都需要通过支撑电路连接和启用外围设备，这些外围设备包括系统主存（DRAM）、ROM或者闪存、系统总线（如PCI总线）以及其他外设，如键盘控制器、串口和IDE设备接口等。这些处理器外围支撑电路通常称为芯片组（chipset），而且往往需要针对一系列处理器进行专门设计。

例如，Pentium M处理器需要型号为855GM的芯片组来支持。855GM芯片组是系统内存、图形设备与处理器之间的主要接口。855GM芯片组已优化为Pentium M处理器的配套芯片组。图3-1说明了处理器与芯片组之间的关系。

请注意描述这些芯片组时常用的一些术语。Intel 855GM芯片组就是一个常被称为北桥（northbridge）芯片的示例，因为它可以直接连接到处理器的高速前端总线（FSB）。另一个用来连接各类I/O以及PCI总线的配套芯片称为南桥（southbridge）芯片。所谓南北只不过是因图中两组芯片的相对位置而形成的约定俗成的叫法。这种硬件体系结构中的南桥芯片实际上是一个I/O

^① Freescale公司现在将G4处理器核称为e600处理器核。

控制器，负责提供I/O接口，包含了以太网、USB、IDE、音频芯片、键盘以及鼠标控制器等，如图3-1所示。

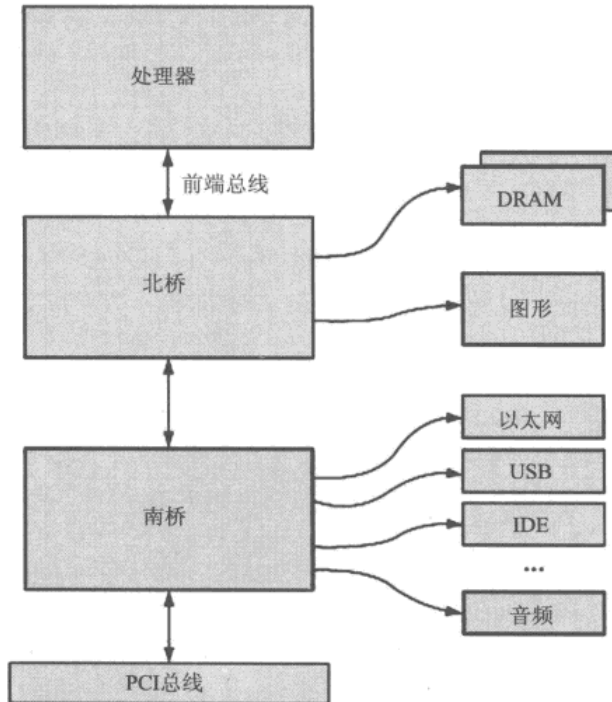


图3-1 处理器与芯片组之间的关系

在PowerPC类型的处理器中，型号为Tundra Tsi110的主机桥接芯片是支持单机PowerPC处理器的典型芯片组。Tsi110芯片组支持单机PowerPC处理器的几种不同类型的接口，它同时可以支持Freescale MPC 74xx系列和IBM PPC 750xx系列处理器。这些处理器使用Tundra芯片可以提供到以下外围设备的接口：

- DDR DRAM，集成化内存控制器；
- 以太网（Tundra芯片组提供4个千兆以太网接口）；
- PCI Express总线（支持两个PCI-E接口）；
- PCI/X（PCI2.3、PCI-X以及Compact PCI [cPCI]）；
- 串口；
- I2C；
- 可编程中断控制器；
- 并口。

有很多芯片组制造商，如威盛科技、Marvell、Tundra、nVidia、Intel等。Marvell和Tundra生产的芯片组主要用于PowerPC，其他芯片组则主要用于Intel体系结构。基于Intel x86、IBM或者

Freescale PowerPC等单机处理器的硬件设计都需要使用配套的芯片组，实现与系统设备之间的接口。

Linux用作嵌入式操作系统的一大优势，是Linux能迅速支持新的芯片组。当前，Linux操作系统已经能够支持上述所有芯片组，并且还支持很多其他芯片组。对于所选的芯片组，用户可以通过Linux源代码和配置工具了解相关信息。

3.2 集成化处理器：片上系统

前一节重点介绍了几种单机处理器。尽管这些处理器能够适用于大量应用（包含一些高功率的处理引擎），但绝大部分嵌入式系统都采用集成化处理器，即片上系统（system on chip, SOC）。目前，有大量的片上系统产品可供选择。本节只选择目前行业内技术比较领先的产品，并分别介绍这些产品的主要特色。与前一节类似，我们只关注Linux提供了良好支持的集成化处理器。

现在有几种主要的处理器体系结构，每一种体系结构都有一些片上系统的例子。PowerPC处理器在网络和电信相关的嵌入式应用中具有传统优势，而MIPS类型的处理器在一些低端的消费类电子产品中占有一席之地^①，ARM处理器则在移动电话中广泛使用。这些表明，主要的体系结构已经广泛使用在嵌入式Linux系统中。从第4章可以了解到，Linux目前支持20多种不同的硬件体系结构。

3.2.1 PowerPC

PowerPC处理器是一种RISC体系结构，由苹果电脑公司、IBM和摩托罗拉的半导体部门（现在摩托罗拉半导体部门已经从摩托罗拉公司分离出来，成立了飞思卡尔半导体公司，即Freescale Semiconductor）共同开发。很多文档都详细描述了PowerPC体系结构，可以先从本章末尾的“参考资料”中罗列的文档入手。

在各种各样的嵌入式产品中几乎都能找到PowerPC处理器的身影，从汽车、消费类电子产品、网络应用设备到最大规模数据及电信交换机，PowerPC是嵌入式应用中最流行的一种体系结构。正是由于PowerPC如此流行，才会存在数家制造商为PowerPC系统提供的大量软硬件解决方案。

3.2.2 AMCC PowerPC

本书后面章节的一些示例是基于AMCC PowerPC 440EP 嵌入式处理器的。440EP处理器是很多连网和通信产品中常见的嵌入式集成化处理器，它具有如下特征：

- 片上DDR SDRAM控制器；
- 集成NAND闪存控制器；
- 具有PCI总线接口；
- 支持双10/100Mbit/s以太网口；
- 片上USB 2.0接口；

^① 这些是作者基于市场的观察所得的个人观点，并不是基于某些统计数据而得出的结论。

- 支持最多4个用户可配置的串口；
- 双I²C控制器；
- 可编程中断控制器；
- 串行外围接口（SPI）控制器；
- 可编程计时器；
- 调试用JTAG接口。

440EP处理器是一款完整的片上系统，图3-2是AMCC PowerPC 440EP处理器的内部组成框图。配合外加的内存芯片和物理I/O硬件，就可以构建一个围绕这个集成化处理器且只需要很少接口电路的完整高端嵌入式系统。

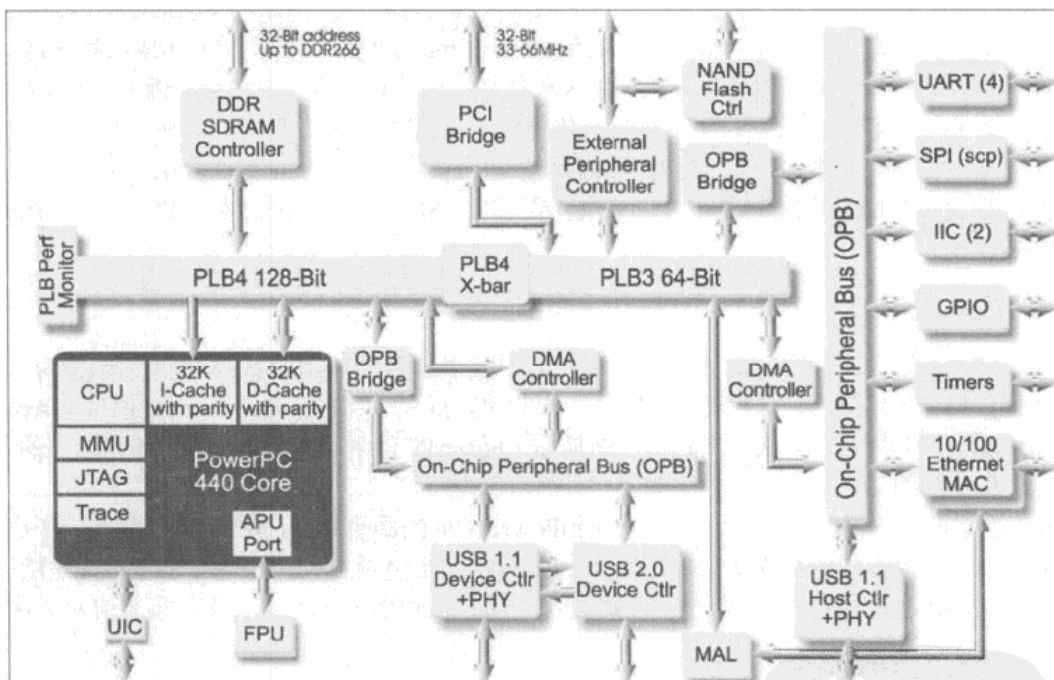


图3-2 AMCC PPC 440EP嵌入式处理器（经AMCC公司许可）

许多制造商都会提供参考硬件平台，便于开发人员考查处理器或其他硬件的功能。第14章和第15章将给出使用AMCC Yosemite评估板来运行的相应示例，该评估板就是由AMCC提供的，包含图3-2所示440EP处理器的参考平台。

PowerPC处理器也提供很多硬件配置用于不同的应用，如图3-2所示，AMCC 440EP处理器为大量常见产品提供了丰富的I/O接口，它只有很少的附加电路。由于440EP处理器集成了浮点运算单元（FPU），因此非常适用于网络设备、通用工业控制系统和直接连网成像设备。

AMCC PowerPC产品线也包含了几种不同的产品，这些产品应用了两个处理器核。基于405核的产品不提供以太网控制器接口，它能够提供的接口或者外围设备包括集成SDRAM控制器、

双UART接口（用于串行通信）、I²C（用于板上的底层管理通信）、集成计数器以及一些通用的I/O端口等。基于AMCC 405核的集成化处理器产品价格公道而且性能优良，非常适用于不需要硬件FPU的应用。

基于AMCC 440核的产品可以提升性能并增加外围电路。例如，在本书一些实例中使用的440EP处理器包含了硬件FPU，而440GX处理器则额外提供了两个10/100/1000 Mbit/s的三速以太网接口（外加两个10/100Mbit/s以太网接口）和可以用于高性能网络设备应用的TCP/IP硬件加速。440SP处理器增加了支持RAID 5/6应用的硬件芯片。所有这些处理器都能够支持Linux操作系统。表3-1中总结了AMCC 405系列处理器的特性，而表3-2中总结了AMCC 440系列处理器的特性。

表3-1 AMCC 405系列处理器特性

特 性	405CR	405EP	405GP	405GPr
内核/主频	PowerPC 405 (133-266) MHz	PowerPC 405 (133-333) MHz	PowerPC 405 (133-266) MHz	PowerPC 405 (266-400) MHz
DRAM控制器	SDRAM/133	SDRAM/133	SDRAM/133	SDRAM/133
以太网 10/100	N	2	1	1
GPIO线	23	32	24	24
UART控制器	2	2	2	2
DMA控制器	4通道	4通道	4通道	4通道
I ² C	Y	Y	Y	Y
PCI主控制器	N	Y	Y	Y
中断控制器	Y	Y	Y	Y

注 详细信息可以查阅AMCC主页www.amcc.com/embedded。

表3-2 AMCC 440系列处理器特性

特 性	440ER	440GP	440GX	440SP
内核/主频	PowerPC 440 (333-667) MHz	PowerPC 440 (400-500) MHz	PowerPC 440 (533-800) MHz	PowerPC 440 (533-667) MHz
DRAM控制器	DDR	DDR	DDR	DDR
以太网 10/100	N	2	1	千兆以太网
千兆以太网	N	N	2	1
GPIO线	64	32	32	32
UART	4	2	2	3
DMA控制器	4通道	4通道	4通道	3通道
I ² C	2	2	2	2
PCI主控制器	Y	PCI-X	PCI-X	3个PCI-X
SPI主控制器	Y	N	N	N
中断控制器	Y	Y	Y	Y

3.2.3 Freescale PowerPC

Freescale公司提供了数量众多的含集成化外设的PowerPC处理器，目前该公司已经调整其PowerPC产品策略，将其划分为3个主要市场：网络设备、汽车电子以及工业控制。Freescale PowerPC处理器目前在网络设备市场中取得了非常可观的成功，其处理器产品线被广泛地应用于各种高低端网络设备。

根据Freescale公司最新发布的信息，该公司目前已经累计供货2亿多颗集成化处理器，这些处理器多数用于通信设备中^①。之所以成功，部分原因是采用了PowerQUICC处理器产品线。PowerQUICC体系结构并不是什么新鲜事物，它已经存在十余年之久。这种处理器以集成了QUICC引擎的PowerPC处理器核为基础（根据Freescale公司的文献，QUICC引擎也叫作通信处理器模块或者CPM）。QUICC引擎也是一种独立的RISC处理器，其目的就是减少主PowerPC处理器核进行通信处理的负荷，这样PowerPC核就可以更加关注于对应用的控制和管理。QUICC引擎尽管比较复杂，但是对于通信外围控制器来说，却提供了非常灵活的手段。

目前，PowerQUICC系列处理器也划分为4大类。为简便起见，下面的讨论中将PowerQUICC产品简写为PQ。

PQ I系列产品包含了最初基于PowerPC处理器开发的PowerQUICC处理器，并由MPC8xx系列处理器产品组成。这些集成化通信处理器产品运行在50 MHz~133 MHz，主要特性由嵌入式PowerPC 8xx核实现。PQ I系列处理器主要用于ATM和以太网边缘设备，例如家用路由器、小型家用办公设备、普通居民住宅用的网关设备、ASDL和有线调制解调器等。

CPM或者QUICC引擎含有两个独特而又强大的通信控制器。SCC（串行通信控制器）是灵活的串行接口，可以实现很多基于串行通信的协议，包括以太网、HDLC/SDLC、AppleTalk、同步或者异步UART、IrDA以及其他一些数据流协议。

SMC（串行管理控制器）也可用于实现类似的串行通信协议，例如ISDN、串行UART、SPI协议等。

将SMC和SCC两种控制器结合起来就能够实现非常灵活的I/O组合，其内部的时分多路复用器更是可以利用这些接口，来实现诸如T1 或者E1类型的I/O。

表3-3列出了部分PQ I系列处理器的主要特性。

表3-3 部分PowerQUICC I系列处理器的产品特性

特 性	MPC850	MPC860	MPC875	MPC885
内核/主频	PowerPC 8xx 最高80MHz	PowerPC 8xx 最高80MHz	PowerPC 8xx 最高133MHz	PowerPC 8xx 最高133MHz
DRAM控制器	Y	Y	Y	Y
USB	Y	N	Y	Y
SPI控制器	Y	N	N	N
I ² C控制器	Y	Y	Y	Y

^① 在Freescale公司网站的Media Center, Press Releases页面上，可以查阅到2005年10月31日发表的一篇通讯。

(续)

特 性	MPC850	MPC860	MPC875	MPC885
SCC控制器	2	4	1	3
SMC控制器	2	2	1	2
安全引擎	N	N	Y	Y
以太网控制器	N	N	2	2

后来开发出了PQ II Freescale PowerPC处理器产品。PQ II产品集成了在Freescale 603e嵌入式处理器核基础上发展而来的G2 PowerPC处理器核。该集成化通信处理器可以运行在133 MHz~450 MHz, 并且提供了多个10/100 Mbit/s以太网接口、安全引擎, 支持ATM和PCI等。PQ II系列产品包含了型号为MPC82xx的处理器产品。

PQ II处理器产品为QUICC引擎增加了两个新的控制器。FCC控制器是进行全双工快速串行通信的控制器, 它支持高速通信, 例如10/100 Mbit/s以太网或者最高能够达到45 Mbit/s的T3/E3通信端口。MCC控制器是进行数据通信的多通道控制器, 每通道具有128 KB×64 KB的数据吞吐能力。

表3-4总结了部分PQ II系列处理器的主要特性。

表3-4 部分PowerQUICC II处理器产品特性

特 性	MPC8250	MPC8260	MPC8272	MPC8280
内核/主频	G2/603e (150-200) MHz	G2/603e (100-300) MHz	G2/603e (266-400) MHz	G2/603e (266-400) MHz
DRAM控制器	Y	Y	Y	Y
USB	N	N	Y	通过SCC4实现
SPI控制器	Y	Y	Y	Y
I ² C控制器	Y	Y	Y	Y
SCC控制器	4	4	3	4
SMC控制器	2	2	2	2
FCC控制器	3	3	2	3
MCC控制器	1	2	0	2

基于Freescale PowerPC e300核(它是G2/603e核心的升级产品), PowerQUICC II Pro系列处理器可以运行在266 MHz~667 MHz, 并且具有支持千兆以太网、DDR SDRAM控制器、PCI、高速USB、安全引擎等特性。PowerQUICC II Pro系列产品包含了MPC83xx系列处理器。PQ II和PQ II Pro系列产品具有非常广泛的应用领域, 例如LAN或者WAN交换机、集线器和网关、PBX系统以及很多类似的具有复杂度和高性能需求的系统。

PowerQUICC II Pro包含三大系列产品, 其中一种不包含QUICC引擎, 而另外两种则包含了QUICC引擎升级版产品。MPC8358E和MPC8360E具有新的通用通信控制器, 支持多种通信协议。

表3-5总结了部分PQ II Pro系列处理器的主要特性。

表3-5 部分PowerQUICC II Pro处理器产品特性

特 性	MPC8343E	MPC8347E	MPC8349E	MPC8360E
内核/主频	e300 (266-400) MHz	e300 (266-667) MHz	e300 (400-667) MHz	e300 (266-667) MHz
DRAM控制器	Y-DDR	Y-DDR	Y-DDR	Y-DDR
USB	Y	2	2	Y
SPI控制器	Y	Y	Y	Y
I ² C控制器	2	2	2	2
以太网10/100/1000	2	2	2	通过UCC实现
UART	2	2	2	2
PCI控制器	Y	Y	Y	Y
安全引擎	Y	Y	Y	Y
UCC控制器	0	0	0	8
MCC控制器	0	0	0	1

PowerQUICC系列处理器产品中的顶级产品就是PQ III系列处理器, 该系列处理器可以运行在600 MHz~1.5 GHz, 以PowerPC e500核为基础, 支持千兆以太网、DDR SDRAM、RapidIO、PCI和PCI-X、ATM、HDLC等。PowerQUICC III系列处理器包括MPC85xx系列产品线。这些处理器主要用于高端应用中, 例如无线网络基站控制器、光纤边缘交换机、中央办公交换机等类似设备。

表3-6总结了部分PQ III系列处理器的主要特性。

表3-6 部分PowerQUICC II Pro处理器产品特性

特 性	MPC8540	MPC8548E	MPC8555E	MPC8560
内核/主频	e500 最高1GHz	e500 最高1.5GHz	e500 最高1GHz	e500 最高1GHz
DRAM控制器	Y-DDR	Y-DDR	Y-DDR	Y-DDR
USB	N	N	通过SCC实现	N
SPI控制器	N	N	Y	Y
I ² C控制器	Y	Y	Y	Y
以太网 10/100	1	千兆以太网	通过SCC实现	通过SCC实现
千兆以太网	2	4	2	2
UART	2	2	2	通过SCC实现
PCI控制器	PCI/PCI-X	PCI/PCI-X	PCI	PCI/PCI-X
RapidIO	Y	Y	Y	Y
安全引擎	N	Y	Y	N
SCC	—	—	3	4
FCC	—	—	2	2
SMC	—	—	2	0
MCC	—	—	0	2

3.2.4 MIPS

你可能会对MIPS类型处理器感到惊奇，因为基于MIPS体系结构的32位处理器已经问世20多年了。MIPS体系结构设计于1981年，由一支斯坦福大学的工程队伍开发完成。领军人物是John Hennessey博士，后来他成立了MIPS Computer System公司。现在，该公司已经改名为MIPS Technology公司，主要工作就是设计并发布MIPS体系结构和处理器核。

很多公司购买了MIPS核的研发许可权，部分产品占据了嵌入式处理器市场非常重要的一席之地。MIPS处理器使用RISC体系结构，支持许多流行产品的32位和64位实现。MIPS处理器广泛使用在很多产品中，从高端产品到消费类电子产品。MIPS处理器应用在很多流行的消费类电子产品中，例如Sony公司的高清晰电视、LinkSys公司无线接入点产品以及Sony公司推出的PS2游戏机等^①。

MIPS Technology公司的网站上一共列出了73家拥有使用MIPS处理器核生产产品的许可权的公司，部分公司是业内响当当的巨头，例如Sony、Texas Instruments、Cisco's Scientific Atlanta（电视机顶盒产品的领头羊）、Motorola等。当然，其中最大也是最成功的生产厂商之一就是Broadcom公司。

3.2.5 Broadcom MIPS

Broadcom公司是片上系统（SOC）解决方案的领先供应商，其业务领域涉及有线电视顶盒、有线调制解调器、HDTV、无线网络、千兆以太网和VoIP系列产品等。Broadcom的SOC产品在上述领域被广泛采纳。之前提及家庭日常生活中已经存在了大量采用Linux操作系统的嵌入式设备，即使你还不知道，这其中也必定有使用Broadcom公司基于MIPS核的片上系统产品。

在2000年，Broadcom公司收购了SiByte公司，从而涉足通信处理器产品市场。目前Broadcom公司能够提供单核、双核甚至四核的处理器，当然，Broadcom公司依然将这些处理器称为SiByte处理器。

单核SiByte处理器包含了BCM1122和BCM1125H两种型号。它们都基于MIPS64处理器核，运行主频在400 MHz~900 MHz。处理器包含DDR SDRAM控制器、10/100 Mbit/s以太网控制器、PCI控制器等片上外设控制器，同时还包含SMBus串行接口、PCMCIA以及两个UART串行端口配置。BCM1125H处理器还提供了一个10/100/1000 Mbit/s三速的以太网控制器。这些处理器的一个最显著的特性就是其低功耗特性，运行在400 MHz的主频下，功耗只有4 W。

双核SiByte处理器包含了BCM1250、BCM1255和BCM1280三种型号，同样基于MIPS64处理器核，运行主频在600 MHz（BCM1250）~1.2 GHz（BCM1255、BCM1280）。这几种双核处理器包括集成化外设控制器，如DDR SDRAM控制器、不同数量的千兆以太网控制器、64位的PCI-X总线接口控制器、SMBus、PCMCIA以及多个UART接口。与单核处理器类似，这些双核处理器也具有显著的低功耗特性，例如BCM1255处理器运行在1 GHz主频时，其功耗只有13 W。

四核SiByte处理器包含了BCM1455和BCM1480两种通信处理器，与前两种SiByte处理器类

^① www.mips.com/content/PressRoom/PressReleases/2003-12-22。

似，该产品也是基于MIPS64处理器核，运行主频在800 MHz~1.2 GHz。该系列处理器内部集成了DDR SDRAM控制器、4个千兆以太网MAC控制器、64位PCI-X主机控制器以及SMBus、PCMCIA、4个串行UART接口等。

表3-7总结了部分Broadcom SiByte处理器的基本特性。

表3-7 部分Broadcom SiByte处理器产品特性

特 性	BCM1125H	BCM1250	BCM1280	BCM1480
内核/主频	SB-1 MIPS64 (400-900) MHz	双SB-1 MIPS64 (600-1000) MHz	双SB-1 MIPS64 (800-1200) MHz	四SB-1 MIPS64 (800-1200) MHz
DRAM控制器	Y-DDR	Y-DDR	Y-DDR	Y-DDR
串行接口	2-55Mbit/s	2-55Mbit/s	4 UART	4 UART
SMBus接口	2	2	2	2
PCMCIA	Y	Y	Y	Y
千兆以太网 (10/100/100) Mbit/s	2	3	4	4
PCI控制器	Y	Y	PCI/PCI-X	PCI/PCI-X
安全引擎	N	N	N	
高速I/O (HyperTransport)	1	1	3	3

3.2.6 AMD MIPS

AMD公司(Advanced Micro Devices Inc)在嵌入式MIPS处理器市场也占据重要一席。在2002年，AMD公司收购了Alchemy Semiconductor公司，从而能够向嵌入式市场提供基于MIPS32核和体系结构的单片集成化SDC。收购后的Alchemy产品线基于流行的MIPS32处理器核，具有相对功耗低、高度系统集成化等特性。

Au1000和Au1100处理器工作主频在266 MHz~500 MHz。两种处理器都集成了SDRAM控制器和独立总线控制器，可以附加到闪存或者PCMCIA等外部设备。表3-8总结了当前几种Alchemy处理器产品线的产品。

表3-8 AMD Alchemy MIPS处理器产品特性

特 性[*]	Au1000	Au1100	Au1200	Au1500	Au1550
内核/主频	MIPS32 (266-500) MHz	MIPS32 (333-500) MHz	MIPS32 (333-500) MHz	MIPS32 (333-500) MHz	MIPS32 (333-500) MHz
DRAM控制器	SDRAM	SDRAM	DDR	SDRAM	DDR
以太网	2	1		2	2
GPIO	32	48	48	39	43
UARTs	4	3	2	2	3
USB1.1	Host+Device	Host+Device	USB2.0	Host+Device	Host+Device

(续)

特性[*]	Au1000	Au1100	Au1200	Au1500	Au1550
AC97音频解 码器	1	1	通过SPC实现	1	通过SPC实现
I ² S	1	1	通过SPC实现		通过SPC实现
SD/MMC	N	2	2	N	N

[*] 其他外设包含IrDA控制器、LCD控制器、2SPC、电源管理、DMA引擎、RTC、摄像头接口、LCD控制器、编解码硬件加速器、PCI主控制器、4 SPC、安全引擎，等等。

3

3.2.7 其他类型的 MIPS

如前所述，在MIPS公司的网站www.mips.com/content/Licensees/ProductCatalog/licensees上能够找到上百个已经发布的MIPS许可。遗憾的是，这里因篇幅所限不能一一列出。大家可以通过MIPS网站查找符合自己需求的MIPS处理器厂商。

例如，ATI Technologies公司使用MIPS核开发了Xilleon机顶盒系列产品的芯片组，Cavium Network的Octeon系列产品使用MIPS64核开发了多核处理器。Integrated Device Technology公司开发了集成化通信处理器Interprise，它也是基于MIPS体系结构的。PMC-Sierra、NEC、Toshiba等公司也开发了基于MIPS核的处理器。对于上述及其他处理器，Linux都提供了良好的支持。

3.2.8 ARM

基于ARM处理器体系结构的产品同样也占据着非常大的消费类电子市场份额。目前有非常多流行且普通的产品使用了ARM处理器核，其中一些比较著名的产品包括Sony PlayStation Portable（便携式PS游戏机）、Apple公司的iPod Nano^①、Nintedo Game Boy Micro和DS产品、TomTom GO 300 GPS、Motorola E680i移动电话（该产品使用了嵌入式Linux）。可以说，基于ARM核的处理器占据着当前数字移动电话的半壁江山，详细信息可以参考ARM公司的文档www.arm.com/micsPDFs/3822.pdf。

ARM处理器体系结构由ARM公司开发并且拥有知识产权，全球的半导体产品制造商都可以选择ARM处理器体系结构开发自己的产品。目前许多全球知名的半导体公司都拥有ARM技术的许可证，并且发布了多种基于某种ARM处理器核的集成化处理器。

3.2.9 TI ARM

Texas Instruments公司使用ARM核开发了OMAP系列集成化处理器。这些处理器包含很多集成化的外围设备，意在作为单一的芯片解决方案，针对各种消费类电子产品，例如手持终端、PDA等类似的多媒体平台。除了常出现在集成化处理器上的接口，例如UART、I²C控制器等，OMAP系列处理器包含了以下更广泛的具有专用目的的接口：

- LCD以及背光控制器；

① 根据ARM公司于2005年圣诞节期间发布的报告。

- 蜂鸣器驱动接口;
- 摄像头接口;
- MMC/SD闪存卡控制器;
- 电池电源硬件管理;
- USB 客户端/主机端接口;
- 无线调制解调器接口逻辑;
- 集成2D或者3D图形加速器;
- 集成安全加速器;
- S-Video输出;
- IrDA控制器;
- DAC, 可以用于电视信号 (PAL/NTSC) 输出;
- 集成DSP, 可以用于视频或者音频处理。

很多流行的手持终端和PDA设备都采用了TI公司开发的OMAP系列处理器, 由于该处理器基于ARM核, 因此已经能够支持Linux操作系统。表3-9对TI OMAP系列处理器中较新的成员特性进行了比较。

表3-9 TI ARM OMAP处理器产品特性

特 性	OMAP1710	OMAP2420	OMAP2430	OMAP3430
内核/主频	ARM926 TEJ 最高200MHz	ARM11 330MHz	ARM1136 330MHz	ARM Cortex A8 550MHz
DRAM控制器	SDRAM	SDRAM	SDRAM	SDRAM
UART	Y	Y	Y	Y
USB	Client+Host	Client+Host	Client+Host	Client+Host
I ² C控制器	Y	Y	Y	Y
MMC-SD接口	Y	Y	Y	Y
键盘控制器	Y	Y	Y	Y
摄像头接口	Y	Y	Y	Y
图形加速功能	2D	2D/3D	2D/3D	Y
集成DSP	TMS320C55x	TMS320C55x	N	N
视频加速硬件	N	IVA (成像视频加 速器)	IVA2 (成像视频加 速器)	IVA2+ (成像视频 加速器)
安全加速	Y	Y	Y	Y
音频解码器	Y	Y	Y	Y
蓝牙和RF调制解 调器接口	Y	Y	Y	Y
LCD控制器	Y	Y	Y	Y
显示控制器	N	PAL/NTSC VGA/QVGA	PAL/NTSC VGA/QVGA	PAL/NTSC VGA/QVGA

3.2.10 Freescale ARM

ARM体系结构的成功，还表现在业内领先的厂商竞相获取ARM技术的许可授权，Freescale公司就是其中之一。该公司采用ARM处理器核开发了i.MX系列处理器。这些基于ARM处理器核的集成化处理器在多媒体消费品设备上获得了成功，如手持游戏终端、PDA、手持终端等。

Freescale公司的ARM系列产品包含了Freescale i.MX21系列产品和i.MX31系列产品。i.MX21产品基于ARM9核进行开发，而i.MX31产品基于ARM11核进行开发。与TI公司开发的OMPA产品类似，Freescale公司开发的片上系统产品也包含了具有多媒体需求的便携式消费类电子产品所需的集成化外设。i.MX21/31系列产品集成了下列接口：

- 图形加速器；
- MPEG-4编码器；
- 小键盘和LCD控制器；
- 摄像头接口；
- 音频多路复用器；
- IrDA红外I/O接口；
- SD/MMC接口；
- 多种扩展I/O，例如PCMCIA、USB、DRAM控制器以及用于串行端口连接的UART等。

3.2.11 Intel ARM XScale

Intel公司也基于ARM v5TE体系结构开发了自己的集成化处理器，将其命名为XScale。XScale处理器根据其应用领域可以分为几大类，表3-10总结了XScale系列处理器的分类（应用类型）。

表3-10 Intel XScale系列处理器

分 类	应 用	处理器实例
应用处理器	手持终端、PDA	PXA27x、PXA29x
I/O处理器	用于数据存储、数据打印、远程数据处理等的高速数据处理	IOP331/332/333
网络处理器	网络以及通信、数据平面处理、快速分组处理，等等	IXP425、IXP465 IXP2350、IXP2855

很多消费类电子产品和网络产品都是使用Intel XScale体系结构处理器进行开发的。一些著名的产品有Garmin公司生产的GPS iQue M5、惠普公司生产的iPAQ、基于Palm处理器的智能手机Treo、Motorola公司的智能手机A760等。Linux操作系统已经全面支持该系列的所有处理器。

在高性能网络设备中也可以找到Intel的网络处理器，在这些设备中需要高速数据吞吐处理。例如深度分组检查、数据加密和解密、分组过滤以及信号处理。这些网络处理器除了包含基本的ARM核以外，还结合了一个或多个处理引擎——叫作网络处理引擎（NPE）。这些NPE专门用于实时的特定数据路径操作。NPE实际上是一个微处理器，可以执行一些微码算法，与ARM处理器核并行完成网络数据的处理。有兴趣的读者可查看Intel公司的网站（www.intel.com）详细了解

XScale系列处理器的技术细节。

3.2.12 其他 ARM

业内有超过100家半导体公司是基于ARM技术开发自己的处理器的，这里不一一列出。很多基于ARM处理器核的产品都专门针对一个特殊应用领域进行开发以适用于特定的市场，例如手持终端市场、数据存储市场、网络处理、汽车电子市场等。这些公司包括Altera、PMC-Sierra、三星电子、飞利浦半导体、富士通等。有兴趣的读者可以到ARM Technologies公司的网站(www.arm.com)查阅相关的详细信息。

3.2.13 其他体系结构

前几节基本上覆盖了在嵌入式Linux操作系统中广泛使用的主要体系结构。不过，Linux操作系统也支持其他一些体系结构。最近的报告显示，Linux操作系统能够支持的体系结构多达25种，既包含了高性能的64位处理器平台也有传统的32位处理器平台，但有些32位平台和64位平台彼此独立，有些系统则有可能存在部分不再支持的接口。

Linux操作系统还能够支持Sun公司的Sparc和Sparc64体系结构、Tensilica公司开发的Xtensa体系结构、NEC的v850体系结构等。稍微花点儿时间浏览一下Linux操作系统内核所能够支持的体系结构，就能了解到Linux操作系统可以支持的所有体系结构。不过得当心，并不是所有内容都及时得到了更新。现在，确实可以选择那些主流的处理平台作为嵌入式系统开发的基础，不过，最好能够紧跟Linux操作系统，特别是嵌入式Linux操作系统供应商的发展步伐。在附录D中，列出了对Linux操作系统应用开发能够有一定帮助的资源列表。

3.3 硬件平台

在系统设计应用中采纳通用的硬件参考平台并不新鲜，PC/104或者VMEbus就是硬件平台的两个例子，并且它们经过了嵌入式系统市场的考验^①。近年来，出现了更多成功的硬件平台，其中包括CompactPCI以及相关的衍生产品。

3.3.1 CompactPCI

CompactPCI (cPCI) 硬件平台是基于PCI电气标准和Eurocard体系规范的。cPCI系统具有下列特性：

- 3U或者6U尺寸的直列插卡形式；
- 基于安全的考虑可以锁闭系统；
- 支持前面板I/O连接或者后面板I/O连接；
- 具有高密度的背板连接；

^① VMEbus实际上并不是一种硬件参考平台，而是基于Eurocard的一种体系规范，并且有多家供应商通过认证并且能够使用该系统标识。

- 叉排电源管脚支持热插拔;
- 多厂商支持;
- 兼容标准的PCI芯片组。

可以从PCI工业计算机制造商联盟 (PCIMG) 的网站www.pcimg.org/compactpci.stm上查阅CompactPCI的相关标准和规范。

3.3.2 ATCA

cPCI获成功后, 其后续者是高性能电信计算体系结构 (Advanced Telecommunication Computing Architecture, ATCA), 它不仅是一种体系结构同时也是一种平台, 该平台是基于PCIMG 3.x系列规范设计开发的。目前很多顶级硬件制造商已经开始开发新的基于ATCA的平台。ATCA平台的主要应用就是电信级产品交换机或者传输设备、高端数据中心服务器以及存储设备。

ATCA平台引领着当前软硬件产品供应商不断前进, 电信或者网络设备市场的大多数大型设备制造商逐步放弃自己设计生产的定制化硬件产品。相应地, 这种趋势在软件平台也很明显, 从操作系统到所谓的中间件软件 (例如高可用协议栈解决方案)。引发这种趋势的主要原因就是成本降低以及产品面市时间的压力。

ATCA是由几个PCIMG规范定义的, 表3-11中列出了这些规范。

表3-11 ATCA PICMG3.x规范

规 范	说 明
PICMG 3.0	机械规范, 包括接插件、供电、制冷和基本系统管理
PICMG 3.1	以太网和光纤通道交换接口
PICMG 3.2	Infiniband交换光纤接口
PICMG 3.3	StarFabric接口
PICMG 3.4	PCIe接口
PICMG 3.5	RapidIO接口

本节介绍的平台已经成为了现在嵌入式Linux操作系统应用的热门话题, 特别是ATCA体系结构, 该行业越来越多地转向COTS技术, ATCA以及Linux操作系统势必在这种行业趋势中扮演着越来越重要的角色。

3.4 小结

- 目前很多单机处理器都支持Linux操作系统, 最流行的莫过于支持IA32/IA64体系结构或PowerPC体系结构。这些单机处理器作为用于构建极高性能的计算引擎的构造块。本章列举了来自于Intel、IBM和Freescall公司的几种处理器。
- 集成化处理器 (即片上系统) 支配着嵌入式Linux操作系统的发展。许多厂商及其一些流行的体系结构用于了嵌入式Linux设备的设计中, 本章同样列举了几个流行的体系结构及其生产商。

- 另一个越来越流行的趋势就是，从选择合适的软硬件平台转向了COTS技术解决方案。目前嵌入式Linux操作系统应用领域较为流行的平台是cPCI和ATCA。

参考资源

32位PowerPC体系结构参考手册：

Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture—Revision 2

Freescale Semiconductor, Inc.

www.freescale.com/files/product/doc/MPCFPE32B.pdf

64位PowerPC体系结构参考手册：

The Programming Environments Manual for 64-Bit Microprocessors—Version 3.0

IBM公司

PowerPC 体系结构简要概述

A Developer's Guide to the POWER Architecture

Brett Olsson, Processor Architect, IBM Corp.

Anthony Marsala, Software Engineer, IBM Corp.

www-128.ibm.com/developerworks/linux/library/l-powarch/

Intel XScale主页

www.intel.com/design/intelxscale/



本章内容

- 背景知识
- Linux内核构造
- 内核构建系统
- 获取Linux内核
- 小结

如果你想要了解有关Linux内核的内部细节，可以找到非常多的图书资料，这些书籍介绍了内核的设计与操作，本章末尾的“参考资源”会涉及一些。但是，很少有从项目角度来介绍内核如何组织和构造的。如果恰好需要为新的嵌入式系统项目增加一些自定义的功能，那该怎么办？如何才能确认哪些文件对于项目所用到的体系结构是不可或缺的呢？

乍一看，理解Linux内核以及为特定平台或应用程序配置内核是一项不可能完成的任务。在目前的Linux内核中，Linux内核源码树包含了20 000多个文件，累积的代码量超过了6百万行。而这些内容仅仅是学习内核的开始，为了构建出一个有用的系统还需要一些工具、一个根文件系统和许多应用程序。

本章主要介绍Linux内核，以及如何组织Linux内核，如何构造Linux源码树。然后介绍组成Linux内核映像的组件，并讨论Linux内核源码树的结构。接着，介绍如何实现内核系统的编译，介绍对内核进行配置的方法以及影响编译过程的配置文件。最后讨论一个完整的嵌入式Linux系统所需的各个组件。

4.1 背景知识

Linus Torvalds编写了Linux操作系统的最初版本，当时他还是芬兰赫尔辛基大学的学生。他从1991年开始工作，当年8月，他在comp.os.minix上留了下面这段著名的通告：

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID: <1991Aug25.205708.9541@klaava.Helsinki.FI>
```

Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki

Hello everybody out there using minix -
I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat(same physical layout of the file-system (due to practical reasons)among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs.
It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-).

从那个最初的版本开始，Linux操作系统已经成为了一种健壮、可靠、具有高端特性的操作系统，足以匹敌那些最好的商业操作系统。有些资料表明，目前半数以上的因特网服务器都基于Linux服务器。众所周知，在线搜索专家Google使用了大量廉价的、运行了具有故障诊断冗余保护的Linux操作系统的PC，以实现它的搜索引擎。

4.1.1 内核的版本

在很多地方都可以轻而易举地得到Linux内核源代码以及其他组件。书店所售Linux图书中附带的光盘就有不同的版本。你也可以从因特网的多个地址下载Linux内核，甚至完整的Linux发行版。Linux内核的官方主页是www.kernel.org。你可能听说过主流代码（mainline source）或者主流内核（mainline kernel），它们实际上都是指在kernel.org网站上能够找到的Linux源码树。

在本书的编写过程中，Linux的版本是2.6。在很早以前的开发者队伍中，从事开发工作的工程师们为了能够区分Linux内核源码树，使用数字来对其进行编号。Linux内核可以分为两大类：一类是专门用于开发的试验版本，另外一类是稳定的产品级版本。Linux操作系统的版本号由主版本号和次版本号组成，最后还有相应的序列号。在2.6版的Linux操作系统之前，如果小版本号数字是偶数，则表明此版本的Linux操作系统是稳定的产品级产品；如果是奇数，则表明此版本的Linux操作系统是用于开发的试验产品，例如：

- Linux 2.4.x ——产品级内核
- Linux 2.5.x ——试验级内核（开发）
- Linux 2.6.x ——产品级内核

目前，Linux 2.6操作系统内核还没有出现新的开发版本分支。顶级Linux源码树由Andrew Morton和Linus Torvalds共同维护，所有新特性、改进以及修订的错误都由其他几名工程师来完成。

其实了解正在使用的Linux内核版本信息是很容易的。在顶级makefile^①的前几行中就包含了当前操作系统的内核版本，例如下面的几行信息，从这几行信息中可以看到，Linux是2.6.14版本的产品级内核。

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 14
EXTRAVERSION =
NAME=Affluent Albatross
```

在这个makefile的后面，还有一些用于生成版本的宏定义，例如：

```
KERNELRELEASE=$(VERSION) . $(PATCHLEVEL) . $(SUBLEVEL) $(EXTRAVERSION)
```

这个宏在内核源码树中被多次使用，用来表示操作系统的内核版本。实际上，操作系统的版本信息使用是非常频繁的，内核开发人员从makefile的版本宏派生了一组宏，在Linux内核资源树的.../include/linux/version.h^②文件中能够找到这些宏。代码清单4-1列出了这些宏。

代码清单4-1 内核头文件：.../include/linux/version.h

```
#define UTS_RELEASE "2.6.14"
#define LINUX_VERSION_CODE 132622
#define KERNEL_VERSION(a,b,c) (((a) << 16) + ((b) << 8) + (c))
```

在Linux操作系统的命令行提示符下使用相应的指令也能够检查当前系统的内核版本信息：

```
$ cat /proc/version
Linux version 2.6.13 (chris@pluto) (gcc version 4.0.0 (DENX ELDK 4.0 4.0.0)) #2
Thu Feb 16 19:30:13 EST 2006
```

关于内核版本最后需要注意的是，用户通过自定义EXTRAVERSION字段，可以很容易跟踪自己开发的内核项目的内核版本信息。例如，如果你正在开发一些新的内核特性，可以像下面这样设置EXTRAVERSION：

```
EXTRAVERSION=-foo
```

当使用cat /proc/version指令时，在命令行窗口中则会看到Linux version 2.6.13-foo，这样就可以很容易区分当前的内核开发进度了。

4.1.2 内核源码库

Linux内核源码的官方网站是www.kernel.org，你可以在这里找到当前和以前版本的Linux内核源码，以及无数的内核补丁。主要的FTP站点ftp.kernel.org也包含了很多子目录，可以一直找到Linux 1.0版本的内核资源。该站点主要关注Linux内核当前正在进行的开发活动。

如果从kernel.org网站下载了近期版本的Linux内核，可能会发现Linux内核源码树具有25种不同的体系结构和子体系结构。造成这种情况的原因之一是不同开发人员对内核系统的修改。如果

① 在这里仅仅简要讨论内核构建系统以及makefile。

② 在本书中，路径信息中的三个连续的点用来表示任意能够访问的Linux操作系统源码树的顶层路径。

每个开发人员都对其关注的部分进行开发并且发布补丁到kernel.org, 那么维护人员就不得不花费巨大的精力来关注相应的变化, 并且管理这些补丁, 那样就无法开展任何有关系统特性方面的开发工作了。正如参加内核开发的人员所言, 已经太忙了。

除了kernel.org网站提供的, 还有几个公共的源码树, 这些资源主要针对特定的体系结构开发。例如, 如果开发人员正在针对MIPS体系结构进行开发, 则适合的内核资源应该在www.linux-mips.org网站上寻找。通常情况下, 对于某种体系结构进行的开发工作最终都要发布到kernel.org网站上, 大多数体系结构开发人员都在努力与主流的内核保持同步, 尽其可能与最新的开发成果保持一致。然而, 很多时候这是一件异常困难的任务, 在主流的内核系统中不见得总能找到合适的补丁, 常常会出现滞后的现象。事实上, 不同体系结构的内核源码树的区别始终存在。

如果你想针对自己特殊的应用找到一个合适的系统内核, 那么, 最好的方法就是获得当前最新的稳定Linux源码树, 然后检查该版本的操作系统是否支持开发项目需要支持的处理器类型, 再后, 搜索Linux内核邮件列表, 找到合适的补丁或者与开发任务相关的一些信息。在邮件列表中还能够找到一些近似的开发补丁以及信息等。

附录D介绍了一些与内核资源库、邮件列表等相关的很好的参考资源。

4.2 Linux 内核构造

接下来几节将分别介绍Linux内核的规划、组织以及构造。结合这些内容, 你就能够轻松地驾驭如此庞大复杂的内核源代码。随着时间的推移, 内核源码树的组织也是几经改进, 特别是在体系结构分支上, 这里面包含了对众多体系结构和专用机器的支持。在本书的编写过程中, 开发人员正在将ppc体系结构和ppc64体系结构合二为一, 组成一个公共的powerpc分支。一旦尘埃落定, 将会为系统带来众多的改进, 其中包括了避免代码重复、文件组织更加完善、分配更加合理, 等等。

4.2.1 顶层资源目录

本书将经常引用“顶层资源目录 (top-level source directory)”这个专有词汇, 指的是Linux内核源码树的最顶层目录。在给定的计算机系统中, 该顶层目录可以位于任意路径中, 而在桌面Linux工作站中, 相应的资源树则一定在/usr/src/linux-x.y.z路径下, 其中x.y.z表示当前Linux内核的版本号。为了简便起见, 我们在本书中使用.../表示顶层内核资源目录。

顶层资源目录包含了众多子目录。(这里忽略了顶层路径下那些非目录的内容, 当然, 也忽略了为了清晰和简便起见而设置的用于进行资源控制的目录。)顶层资源目录下的子目录包括:

arch	crypto	Documentation	drivers	fs	include
Init	ipc	kernel	lib	mm	net
scripts	security	sound	usr		

通常这些子目录还包含了其他几层子目录, 里面包含了源代码、makefile以及配置文件。所有Linux内核源码树中最大的分支就是那个.../drivers路径下的内容, 这里面包含了支持以太网卡、USB控制器以及Linux内核所支持的所有硬件设备的驱动程序源代码。很容易想到, 第二

大的分支就是.../arch子目录，它包含了20多种不同类型的处理器体系结构的支持文件。

除了这些子目录，在顶层资源目录下还能够找到一些额外的文件，包括顶层makefile、隐藏的配置文件（.config文件，4.3.1节中介绍），以及一些与内核编译不甚相干的信息文件。最后，当Linux内核完成编译后，还能够顶层资源目录下找到两个非常重要的文件：System.map文件以及vmlinux文件，这些内容将在后续章节中分别介绍。

4.2.2 编译内核

理解Linux这样庞大的软件是一项非常艰巨的任务，也正是由于其过分庞大，很难一下找到当前发生的问题究竟是由哪一行代码引起的。Linux操作系统具有多线程抢占式任务环境的特性，这更增加了分析系统的复杂程度。事实上，准确地找到系统入口点——也就是当前Linux内核的第一行执行代码，具有极强的挑战性。理解大二进制映像的结构较为有用的一种方法，是检查操作系统内核的构建组件。

内核构建系统的输出将产生几个通用文件，以及一个或多个特定体系结构的二进制模块，构建通用文件往往与其体系结构无关。前面曾经提及了两个通用文件System.map和vmlinux。前者对于进行操作系统内核调试以及某些特别的兴趣爱好需求很有用处，里面包含了比较简单易懂的内核符号列表及其相应的地址。后者是一个针对特定体系结构的ELF文件^①，是一个可执行文件。对于每个体系结构，它是由位于内核顶层目录的makefile在编译过程中生成的。如果在内核编译过程中使用了一些符号式内核调试信息，则这些信息将包含在vmlinux映像中。在实际应用中，尽管它是一个可执行的ELF文件，但它永远也不会启动过程中直接被调用，这一点你很快就会看到。

代码清单4-2是利用make构建操作系统内核得到的部分输出信息，这里编译的操作系统内核将运行在ARM XScale体系结构下。操作系统内核源码树根据ADI Engineering Coyote开发板进行配置，该开发板基于Intel XP 425网络处理器，具体的配置指令为：

```
make ARCH=arm CROSS_COMPILE=xscale_be- ixp4xx_defconfig
```

这个命令不能构建内核，它负责针对XScale处理器体系结构对内核源码树进行配置，并且提供了一个默认的初始化配置文件，针对这个体系结构和处理器完成编译配置。在命令中，指定了默认的配置文件ixp4xx_defconfig，并且根据这个文件构建了新的用于完成最终编译的内核配置文件。针对处理器编译配置的更多讨论将在4.3节详细介绍。

代码清单4-2演示了构建内核的命令，但是只有整个构建过程的前几行以及最后几行输出列在了这里，供大家学习讨论。

代码清单4-2 内核构建的输出

```
$ make ARCH=arm CROSS_COMPILE=xscale_be- zImage
CHK      include/linux/version.h
HOSTCC   scripts/basic/fixdep
```

^① ELF 是 Executable and Linking Format 的缩写，是二进制可执行文件的事实标准格式。

```

. <hundreds of lines of output omitted here>
.
LD      vmlinux
SYSMAP  System.map
SYSMAP  .tmp_System.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
AS      arch/arm/boot/compressed/head.o
GZIP    arch/arm/boot/compressed/piggy.gz
AS      arch/arm/boot/compressed/piggy.o
CC      arch/arm/boot/compressed/misc.o
AS      arch/arm/boot/compressed/head-xscale.o
AS      arch/arm/boot/compressed/big-endian.o
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
Building modules, stage 2.
...

```

首先需要注意的就是进行内核编译的那条make指令。在该指令中，制定了内核编译的目标体系结构（ARCH=arm）和相应的交叉编译工具链（CROSS_COMPILE=xscale_be-），有了这两个命令行参数，make指令将使用XScale交叉编译工具构建内核映像，并使用内核源码树的特定于arm的分支来构建体系结构相关的部分，同时在编译命令中指定了目标zImage，这个目标针对很多体系结构都比较常见，将在第5章中详细介绍。

接下来需要关注的事情是在实际编译步骤当中使用的真正的命令，这些命令被某些助记符所替代或者隐藏了起来。这么做的目的实际很简单，就是尽可能简化编译过程的输出，让开发人员尽可能关注编译过程中生成的中间文件，特别是编译器输出的一些警告信息。在早期的内核源码树中，内核编译或者链接命令要向控制台详细输出每一步骤，往往需要几行才能将一步输出表示清楚，结果，整个内核编译过程变得非常难读，编译器输出的警告信息被淹没在了繁杂的输出信息中，无法引起足够的关注。在新的版本中，利用助记符隐藏了一些编译细节，这样，编译过程中出现的任何异常情况都很容易被发现。如果需要在编译过程中查看编译器向命令行窗口输出的所有详细信息，则需要在make指令的命令行中，使用v=1命令行参数。

代码清单4-2忽略了大部分内核编译过程中控制台的输出信息，主要是便于大家清楚地了解整个编译的基本过程（实际的编译过程中，要调用超过900个不同的编译、链接指令，如果全部列出，则要占据很长的篇幅）。当所有的中间文件以及库文件都完成编译链接之后，所有文件都将被放置于一个非常大的ELF目标文件中，这个文件就是vmlinux。虽然这个文件根据不同的体系结构会有所不同，但是vmlinux是最常见的一种目标文件，能够支持所有Linux操作系统的体系结构。

4.2.3 严格意义上的内核：vmlinux

注意代码清单4-2中的下面这行：

```
LD arch/arm/boot/compressed/vmlinux
```

vmlinux实际上就是严格意义上的Linux内核，它是一个完全独立运行的单一操作系统内核映像。所有的外部引用信息都保存在vmlinux二进制文件中，当需要执行该内核的上下文时（通过引导装入程序启动Linux内核时），vmlinux内核二进制映像将启动相应的硬件，并且执行一个完整功能的内核。

请大家牢记一点，如果需要充分了解一个系统，最好是从了解其部分或者局部入手，首先看看vmlinux内核映像的基本构造。代码清单4-3列举了在构建内核阶段，生成vmlinux ELF对象文件时，系统控制台的输出信息。为了更可读，在每一行后面增加了“\”表示一行的结束，否则，生成vmlinux映像文件在整个构建过程中仅仅是简单的一个步骤，就像在代码清单4-2中的那样。如果通过手工编译内核，而不是批量编译，则代码清单4-3中的信息就是在命令行窗口中进行链接的所有指令。

代码清单4-3 链接阶段：vmlinux

```
xscale_be-ld -EB -p --no-undefined -X -o vmlinux \
-T arch/arm/kernel/vmlinux.lds \
arch/arm/kernel/head.o \
arch/arm/kernel/init_task.o \
init/built-in.o \
--start-group \
usr/built-in.o \
arch/arm/kernel/built-in.o \
arch/arm/mm/built-in.o \
arch/arm/common/built-in.o \
arch/arm/mach-ixp4xx/built-in.o \
arch/arm/nwfpe/built-in.o \
kernel/built-in.o \
mm/built-in.o \
fs/built-in.o \
ipc/built-in.o \
security/built-in.o \
crypto/built-in.o \
lib/lib.a \
arch/arm/lib/lib.a \
lib/built-in.o \
arch/arm/lib/built-in.o \
drivers/built-in.o \
sound/built-in.o \
net/built-in.o \
--end-group \
.tmp_kallsyms2.o
```

4.2.4 内核映像组件

通过代码清单4-3可以看到，vmlinux映像是由几个二进制映像文件组成的。不过到目前为止，了解每个组件的具体功能还不是特别重要。现在重要的是，大家必须了解操作系统的内核都是由哪些二进制组件组成的。在代码清单4-3的文本中，链接指令的第一行定义了输出文件（-o vmlinux），而第二行定义了链接器脚本文件（-T vmlinux.lds），这个文件将决定内核二进制

文件将如何进行最后的链接工作^①。

从代码清单4-3第3行开始，列举了构成最终二进制映像文件的所有对象模块。注意第一个被引入的模块叫作head.o，这个文件由/arch/arm/kernel/head.S文件生成，它是一个汇编语言的源代码文件，针对特定的体系结构用于完成低层次的内核初始化工作。如果搜索并察看一下被内核执行的源文件代码第一行，读者就能够理解从这里入手的道理，其实链接阶段创建的二进制映像文件的第一行就是这个文件的第一行代码。第5章将详细介绍内核初始化的细节。

接下来的对象模块是init_task.o，它将创建内核所必需的初始化线程以及任务结构，紧随在这个对象模块后面是一系列的初始化模块集合，每个都有共同的名称built-in.o。不过，你可能已经注意到，每个built-in.o对象文件前面都有特定的内核资源路径，这样就可以确定这些built-in.o对象文件是来自于内核的不同组件。这些都是二进制对象文件，将被包含于内核映像文件中。下面将利用图表来更清晰地说明这个问题。

图4-1说明了vmlinux映像文件的构成，包括了内核链接阶段每一行所包含的内容。受篇幅所限，无法扩展，但是依然能够从中看出每个功能模块相对的尺寸。

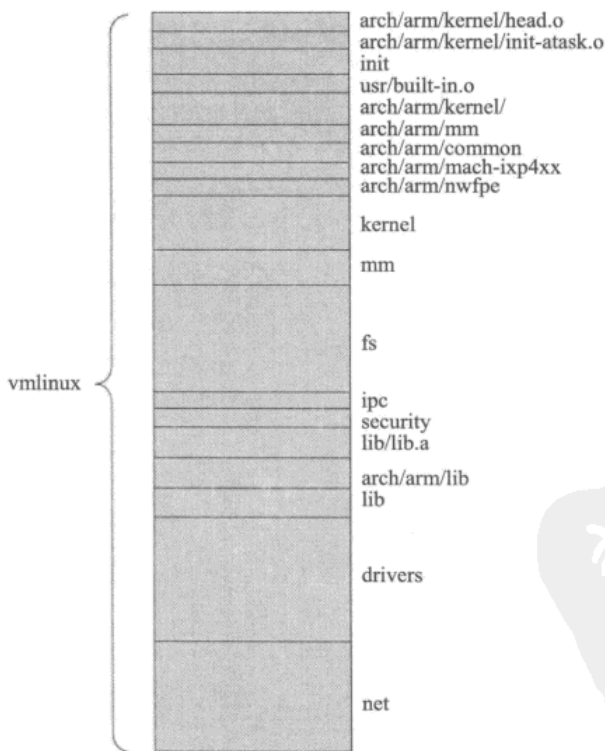


图4-1 vmlinux映像文件组件

^① 链接器脚本文件具有特定的语法规则，在GNU链接器的文档中对此有详细的描述。

从图中可以发现，在所有的二进制文件组件中，最大的3个就是文件系统代码、网络代码以及所有内置驱动程序模块。如果把内核代码和体系结构内核代码组合在一起，则这就是第二大的内核二进制组件。这里包含了任务调度、进程和线程管理、时间管理等各种内核系统功能。一般的，内核还需要包括一些对特定硬件体系结构专门定义的功能，例如底层的上下文切换、硬件层中断和时钟的处理、处理器异常处理等，这些内容在.../arch/arm/kernel文件夹中可以找到。

需要注意一点，这里举出的内核编译实例针对的是某个特定的硬件体系结构，前面曾经提及，此处针对ARM Xscale体系结构进行内核编译，具体一些，这里使用的硬件是ADI Engineering公司声称的具有Intel IXP425网络处理器的参考板，图4-1也指出了针对此硬件的二进制组件arch/arm/mach-ixp4xx。每一种体系结构和机器类型（处理器或者参考板）都多多少少有些许的差别，这样就引起了在内核系统的相应的差别，在vmlinux映像文件中自然也反映出相应的区别。不过如果读者能够理解在本节中所举示例，应该能够举一反三，比较容易地理解其他硬件体系结构的内核系统。

为了便于大家理解内核源码树中每个组件的功能，表4-1列举了各个组件并给出了简短的说明，这些组件就是构成vmlinux内核映像的组件。

表4-1 vmlinux映像中所包含的组件描述

组 件	描 述
arch/arm/kernel/head.o	内核系统针对特定体系结构的启动代码
init_task.o	线程初始化以及任务结构的初始化模块
init/built-in.o	主内核初始化代码，第5章中详细介绍
usr/built-in.o	内置initramfs映像，第5章中详细介绍
arch/arm/kernel/built-in.o	特定体系结构的内核代码
arch/arm/mm/built-in.o	特定体系结构的内存管理代码
arch/arm/common/built-in.o	特定体系结构的通用代码，针对不同体系结构代码不同
arch/arm/mach-ixp4xx/built-in.o	针对特定计算机的代码，通常实现初始化功能
arch/arm/nwfpe/built-in.o	针对特定体系结构的浮点枚举计算代码
kernel/built-in.o	内核的通用代码
mm/built-in.o	内核的内存管理代码
ipc/built-in.o	内部处理通信，例如SysV IPC
security/built-in.o	Linux内核安全模块
lib/lib.a	各种有用的辅助功能函数集合
arch/arm/lib/lib.a	特定体系结构的通用应用库，依体系结构不同而不同
lib/built-in.o	通用内核辅助函数
drivers/built-in.o	所有必要的内建驱动程序，可加载模块
sound/built-in.o	声卡的驱动程序模块
net/built-in.o	Linux操作系统的网络模块
.tmp_kallsyms2.o	符号表

当说起内核映像的时候，往往指的是vmlinux映像文件。正如前文所述，极少有哪一种系统

平台会直接启动这个映像文件，而且这个文件几乎总是被压缩保存的。系统的引导文件总是需要对这个文件进行解压缩。很多平台都需要一些不同类型的接入点以便实现解压缩的操作，在第5章中，将介绍这些映像文件如何针对不同的体系结构、计算机类型、引导装入程序以及启动过程的需要而进行相应的组件打包。

4.2.5 子目录结构

至此，有关内核构建系统如何进行内核映像编译的内容已经介绍完毕，最后简要看一下内核子目录，代码清单4-4中列出了mach-ixp425内核的子目录，这些路径都保存在内核源码树.../arch/arm体系结构分支下。

代码清单4-4 内核子目录

```
$ ls -l linux-2.6/arch/arm/mach-ixp425
total 92
-rw-rw-r-- 1 chris chris 11892 Oct 10 14:53 built-in.o
-rw-rw-r-- 1 chris chris 6924 Sep 29 15:39 common.c
-rw-rw-r-- 1 chris chris 3525 Oct 10 14:53 common.o
-rw-rw-r-- 1 chris chris 13062 Sep 29 15:39 common-pci.c
-rw-rw-r-- 1 chris chris 7504 Oct 10 14:53 common-pci.o
-rw-rw-r-- 1 chris chris 1728 Sep 29 15:39 coyote-pci.c
-rw-rw-r-- 1 chris chris 1572 Oct 10 14:53 coyote-pci.o
-rw-rw-r-- 1 chris chris 2118 Sep 29 15:39 coyote-setup.c
-rw-rw-r-- 1 chris chris 2180 Oct 10 14:53 coyote-setup.o
-rw-rw-r-- 1 chris chris 2042 Sep 29 15:39 ixdp425-pci.c
-rw-rw-r-- 1 chris chris 3656 Sep 29 15:39 ixdp425-setup.c
-rw-rw-r-- 1 chris chris 2761 Sep 29 15:39 Kconfig
-rw-rw-r-- 1 chris chris 259 Sep 29 15:39 Makefile
-rw-rw-r-- 1 chris chris 3102 Sep 29 15:39 prpmc1100-pci.c
```

代码清单4-4所列出的内核路径是很多内核系统都会包含的一些常见内容，这些内容可以在Makefile和Kconfig中找到。这两个文件将用于完成内核的配置以及编译过程，在后面的小节中将介绍这方面的内容。

4.3 内核构建系统

Linux内核的编译与配置机制相当复杂，如果把它看作一个软件工程项目，那么这个软件足足包含了超过6百万行源代码！本节将重点介绍Linux操作系统的内核构建系统，对于开发人员来说需要定制这个内核构建系统来完成系统编译工作。

快速浏览一下当前的Linux内核，你会发现在整个内核源码树中，包含了超过800个makefile^①，这看上去是多么庞大的数字啊！不过，当你理解了整个编译系统的结构和工作过程之后，这个数字看上去就没有那么可怕了。从Linux 2.4版本之后，Linux内核构建机制已经有了显著的改进，相对于你可能比较熟悉的旧版本内核构建机制，在新版本的内核构建机制中提供了Kbuild系统，

① 并不是所有makefile直接影响系统内核编译过程，例如用于说明的文档文件。

这是一个非常显著的改进。本节只介绍较新的内核以及Kbuild编译系统。

4.3.1 .config 文件

如前所述，.config文件包含了编译Linux内核映像的一系列配置脚本。你必须在这个.config文件上花费相当多的时间，它是进行操作系统内核编译的起点，特别是针对某种嵌入式平台完成Linux内核编译的工作。我们可以找到几种不同的编辑器，有基于文本的也有图形界面的，利用这些编辑器编写好的配置脚本文件都叫作.config，这个文件被放置于Linux操作系统资源路径的顶层，用于驱动整个系统内核的构建工作。

正是由于在内核编译配置文件上花费了很多时间和精力，所以一个很重要的问题就是如何保护相应的配置文件。Linux操作系统中存在若干make指令，能够在不给出任何警告的情况下就删除配置文件。最常用的make指令是make mrproper。这条指令将使内核源码树返回到最原始的未曾配置过的状态，在编译的过程中，将删除所有源码树包含的配置文件，当然，也会把辛辛苦苦编写的.config文件一起删除了。

众所周知，在Linux操作系统中，文件名增加一个点作为前缀，则该文件就是非常重要的文件，需要被隐藏起来，但是这样也带来了相当的麻烦，特别是当前的软件工程项目是由多名开发人员完成的时候。如果一个人没有及时备份.config文件，而执行了指令make mrproper，则会不得不陷入失去劳动成果的痛苦中。

.config文件格式的定义比较简单，在代码清单4-5中列出了当前Linux操作系统所包含的.config文件的片段。

代码清单4-5 Linux 2.6 .config文件的片段

```
...
# USB support
#
CONFIG_USB=m
# CONFIG_USB_DEBUG is not set

# Miscellaneous USB options
#
CONFIG_USB_DEVICEFS=y
# CONFIG_USB_BANDWIDTH is not set
# CONFIG_USB_DYNAMIC_MINORS is not set

# USB Host Controller Drivers
#
CONFIG_USB_EHCI_HCD=m
# CONFIG_USB_EHCI_SPLIT_ISO is not set
# CONFIG_USB_EHCI_ROOT_HUB_TT is not set
CONFIG_USB_OHCI_HCD=m
CONFIG_USB_UHCI_HCD=m
...
```

理解.config文件的基础是适当了解一些有关Linux内核的基础知识。Linux是一个单一体系

结构的操作系统，整个操作系统的内核在一次编译链接过程中完成，并且会生成一个静态的可执行系统文件。但是，系统也可以逐步编译并且增量链接^①一组源文件而生成单一的目标模块，这样才能够动态地将一些特性引入到运行的内核系统中。在进行系统驱动程序开发的时候经常采用这种方法，以便支持更多的系统硬件平台。在Linux操作系统中，这些目标模块称为“可加载模块（loadable module）”。内核启动以后，特定的应用程序把这些可加载模块依次插入到正在运行的内核中。

有了这些预备知识后，我们再看看代码清单4-5中所给出的.config文件片段。列表中的配置文件片段针对系统USB设备进行了配置。第一个配置选项CONFIG_USB=m，声明了内核配置的USB子系统需要被编译并且生成动态可加载模块（=m），这些模块将在内核启动之后再动态加载进来。另外一种选项就是可以设置为=y，该选项将使内核系统编译USB子系统并且静态加载使其成为Linux内核的一部分。相应的模块将进入../drivers/built-in.o文件中，之前的代码清单4-3以及图4-1中都解读了该文件。聪明的读者可能已经发现，如果相应的驱动程序模块被编译链接成为可动态加载的模块，则代码不会保存在vmlinux内核中，而是作为一个独立的模块存在，这个模块是可以动态加载的，将在操作系统内核启动后被插入到运行的内核中。

注意代码清单4-5中CONFIG_USB_DEVICEFS=y声明，这个配置选项的行为不大相同。在这里，USB_DEVICEFS（简短起见仅表示配置选项）并不是独立应用的模块，而是某种特性在USB设备使用时被设置为enable或者disable。这个选项不一定是模块所必需的内容，因此不会被编译到内核vmlinux中；而对应地，这个选项将允许一个或者若干个系统特性，作为一种附加的对象模块而引入，从而影响所有系统USB设备模块。通常情况下，读者可以参考配置文件编辑器的帮助信息或者配置文件编辑器所提供的体系框架，深入理解此问题。

4.3.2 配置编辑器

早期Linux内核利用简单的命令行指令完成其配置工作。尽管配置参数不是很多，但是利用命令行进行操作系统配置工作也是非常麻烦的。在现在的系统中，利用命令行进行系统配置的形式依然被保留下来，但是这种方式异常乏味无聊，已经很少使用。因为，在现在的操作系统内核配置工作中，需要从命令行完成600多个参数的配置，相当于回答600多个繁琐的问题，开发人员要在脚本中依次键入相应的选项以及对应的参数值。一旦出现了错误，前面所进行的工作无法恢复，必须从头再来。想象一下，如果不巧在输入第599个选项时出现了错误，将是一件多么令人抓狂的事情啊！

在某些情况下，对于嵌入式系统的Linux内核编译工作无法在图形化的方式下进行，利用命令行指令进行内核的配置工作是无法避免的，不过，依然能够找到一些方法来规避命令行带来的不便之处。

内核配置子系统可以使用多种图形化界面的工具，在最新的Linux内核发行版中包含了10个

^① 所谓增量式链接是一种特别的编译链接技术，利用这种技术能够将不同的模块分别链接到当前的目标模块中，尽管当前的模块中可能存在一些人为定义的符号标识，但是不会引起任何错误，相应的符号标识将在下一次链接阶段被正确识别。

不同的配置目标，在这里小结如下，后面的文本说明可以利用`make help`指令得到：

- ☐ `config`——使用命令程序更新当前配置信息。
- ☐ `menuconfig`——使用菜单程序更新当前配置信息。
- ☐ `xconfig`——使用QT前端更新当前配置信息。
- ☐ `gconfig`——使用GTK前端更新当前配置信息。
- ☐ `oldconfig`——使用`.config`作为基础更新当前配置信息。
- ☐ `randconfig`——全新的配置，其中包含了所有选项的随机答案。
- ☐ `defconfig`——全新的配置，其中包含了所有选项的默认答案。
- ☐ `allmodconfig`——全新的配置，在适当的时候会选择模块。
- ☐ `allyesconfig`——全新的配置，所有选项都接受。
- ☐ `allnoconfig`——全新的最小配置信息。

前4个`makefile`配置目标执行相应的配置文件编辑器，正如后面的文本说明的那样。受篇幅所限，本节将集中讨论基于GTK开发的图形化界面工具。其实，利用不同的图形化开发工具完成配置文件选项的编辑得到的结果是一致的。

如果要执行相应的图形化配置文件编辑器，就在顶层内核路径下，在命令行中键入指令`make gconfig`^①。如图4-2所示，运行`gconfig`时将提供顶层配置选项菜单。基于该工具，每个内核编译选项都可以被访问修改，从而生成自定义的配置文件。

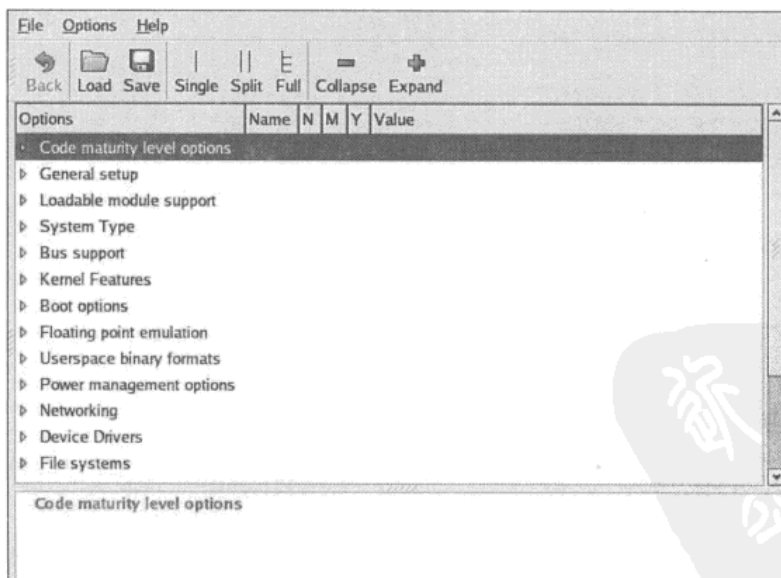


图4-2 顶层内核配置工具

退出配置文件编辑器时，会提示用户对配置文件的修改进行保存。如果选择保存修改，则全

^① 读者可以尝试使用其他的配置工具，例如键入指令`make xconfig`或者`make menuconfig`等。

局配置文件.config将被更新（或者被创建，如果该文件不存在的话）。如前所述，这个.config文件将通过顶层makefile驱动整个系统内核的编译工作。当然，需要在相应的makefile中编写include指令读入相应的.config文件。

大多数内核软件模块也需要直接从.config文件中读取配置信息。在整个编译过程中，.config文件将被处理生成一个C语言头文件，这个文件在.../include/linux目录下能够找到，文件名为autoconf.h。这个文件是一个自动生成的文件，永远也不要直接编辑，因为如果编辑了该头文件，则运行配置文件编辑器之后，头文件中的修改信息将会丢失。大多数内核源文件都要引入该头文件，请读者自行查看一下源文件中的#include定义。代码清单4-6是自动生成的autoconf.h文件的片段，这个片段对应了前面USB子系统配置的示例。要注意在代码清单4-5的.config文件片段中出现的若干选项，在autoconf.h头文件中都有相对应的表示。通过这个例子，大家应该能够理解内核源文件使用系统内核配置文件的基本方法。

代码清单4-6 Linux autoconf.h文件片段

```
/*
 * USB support
 */
#define CONFIG_USB_MODULE 1
#undef CONFIG_USB_DEBUG

/*
 * Miscellaneous USB options
 */
#define CONFIG_USB_DEVICEFS 1
#undef CONFIG_USB_BANDWIDTH
#undef CONFIG_USB_DYNAMIC_MINORS

/*
 * USB Host Controller Drivers
 */
#define CONFIG_USB_EHCI_HCD_MODULE 1
#undef CONFIG_USB_EHCI_SPLIT_ISO
#undef CONFIG_USB_EHCI_ROOT_HUB_TT
#define CONFIG_USB_OHCI_HCD_MODULE 1
#define CONFIG_USB_UHCI_HCD_MODULE 1
```

至此，如果读者还没能理解本节的内容，则请在Linux内核资源路径的顶层下执行make gconfig指令，然后在相应的工具中仔细查看庞大的配置选项分支以及对应分支选项的内容。作为Linux操作系统的开发人员，可以自由修改这些选项。如果不确认自己的修改对Linux内核有怎样的影响，则可以在退出配置工具时不保存修改，尽管对应的修改信息可能会丢失，但是可以保证能够安全浏览其内容而不对内核配置进行任何定义^①。许多配置选项都具有对应的说明解释文本，仔细阅读一下，对理解不同的配置选项会非常有帮助。

① 最好的方法，还是对.config进行备份然后再来修改。

4.3.3 makefile 的目标

如果读者在Linux操作系统资源路径的顶层下执行指令`make help`，就能看到相应的目标列表，这些目标就是利用当前源码树生成的不同目标。最常用的方法就是在使用`make`指令的时候不指定任何目标。这样将生成内核ELF文件`vmlinux`，它对应于目标体系结构下的二进制内核映像文件（例如x86体系结构下的`bzImage`映像文件）。在不指定任何目标的时候，`make`指令将根据配置文件实现所有设备驱动程序模块（可动态加载的模块）的编译链接工作。

很多体系结构或者机器类型都需要特定的二进制目标，以适应其特殊的体系结构和引导装入程序。最常见的一种体系结构所对应的内核目标是`zImage`，在大多数体系结构下，这个二进制映像文件就是默认的文件，能够被加载并运行在对应目标的嵌入式系统中。对于初学者比较常见的一个错误，就是在`make`指令中指定`bzImage`目标。需要牢记，`bzImage`目标专门针对x86/PC体系结构，`bzImage`文件不是一个`bzip2`压缩文件，它是一个很大的`zImage`文件。读者不需要仔细了解以前的PC硬件体系结构细节，而只需要了解`bzImage`目标仅适用于PC机兼容系统，它们都具有符合工业标准的BIOS。

代码清单4-7给出了在命令行中键入`make help`之后得到的输出。读者可以从该代码中充分了解当前Linux内核支持了多少种不同的编译目标。在每个列出的目标后面都有简要的说明文本。需要强调一点，即便是`help`也是`make`指令的目标之一，对于不同的体系结构也有不同的结果。读者可以在不同的体系架构下的Linux操作系统中得到不同的目标列表，代码清单4-7中得到的是在ARM体系结构下的结果。

代码清单4-7 makefile的目标

```
$ make ARCH=arm help
Cleaning targets:
  clean          - remove most generated files but keep the config
  mrproper       - remove all generated files + config + various backup files

Configuration targets:
  config         - Update current config utilising a line-oriented program
  menuconfig     - Update current config utilising a menu based program
  xconfig        - Update current config utilising a QT based front-end
  gconfig        - Update current config utilising a GTK based front-end
  oldconfig      - Update current config utilising a provided .config as base
  randconfig     - New config with random answer to all options
  defconfig      - New config with default answer to all options
  allmodconfig   - New config selecting modules when possible
  allyesconfig   - New config where all options are accepted with yes
  allnoconfig    - New minimal config

Other generic targets:
  all            - Build all targets marked with [*]
  * vmlinux      - Build the bare kernel
  * modules      - Build all modules
  modules_install - Install all modules
  dir/           - Build all files in dir and below
  dir/file.[ois] - Build specified target only
```

```

dir/file.ko      - Build module including final link
rpm              - Build a kernel as an RPM package
tags/TAGS        - Generate tags file for editors
cscope           - Generate cscope index
kernelrelease    - Output the release version string

```

Static analysers

```

buildcheck      - List dangling references to vmlinux discarded sections and
                  init sections from non-init sections
checkstack      - Generate a list of stack hogs
namespacecheck   - Name space analysis on compiled kernel

```

Kernel packaging:

```

rpm-pkg         - Build the kernel as an RPM package
binrpm-pkg      - Build an rpm package containing the compiled kernel and
                  modules
deb-pkg         - Build the kernel as an deb package
tar-pkg         - Build the kernel as an uncompressed tarball
targz-pkg       - Build the kernel as a gzip compressed tarball
tarbz2-pkg      - Build the kernel as a bzip2 compressed tarball

```

Documentation targets:

```

Linux kernel internal documentation in different formats:
xmldocs (XML DocBook), psdocs (Postscript), pdfdocs (PDF)
htmldocs (HTML), mandocs (man pages, use installmandocs to install)

```

Architecture specific targets (arm):

```

* zImage        - Compressed kernel image (arch/arm/boot/zImage)
Image           - Uncompressed kernel image (arch/arm/boot/Image)
* xipImage      - XIP kernel image, if configured (arch/arm/boot/xipImage)
bootpImage      - Combined zImage and initial RAM disk
                  (supply initrd image via make variable INITRD=<path>)
install         - Install uncompressed kernel
zinstall        - Install compressed kernel
                  Install using (your) ~/bin/installkernel or
                  (distribution) /sbin/installkernel or
                  install to $(INSTALL_PATH) and run lilo

assabet_defconfig - Build for assabet
badge4_defconfig  - Build for badge4
bast_defconfig    - Build for bast
cerfcube_defconfig - Build for cerfcube
clps7500_defconfig - Build for clps7500
collie_defconfig  - Build for collie
corgi_defconfig   - Build for corgi
ebsa110_defconfig - Build for ebsa110
edb7211_defconfig - Build for edb7211
enp2611_defconfig - Build for enp2611
ep80219_defconfig - Build for ep80219
epxa10db_defconfig - Build for epxa10db
footbridge_defconfig - Build for footbridge
fortunet_defconfig - Build for fortunet
h3600_defconfig   - Build for h3600
h7201_defconfig   - Build for h7201

```



h7202_defconfig	- Build for h7202
hackkit_defconfig	- Build for hackkit
integrator_defconfig	- Build for integrator
iq31244_defconfig	- Build for iq31244
iq80321_defconfig	- Build for iq80321
iq80331_defconfig	- Build for iq80331
iq80332_defconfig	- Build for iq80332
ixdp2400_defconfig	- Build for ixdp2400
ixdp2401_defconfig	- Build for ixdp2401
ixdp2800_defconfig	- Build for ixdp2800
ixdp2801_defconfig	- Build for ixdp2801
ixp4xx_defconfig	- Build for ixp4xx
jornada720_defconfig	- Build for jornada720
lart_defconfig	- Build for lart
lpd7a400_defconfig	- Build for lpd7a400
lpd7a404_defconfig	- Build for lpd7a404
lubbock_defconfig	- Build for lubbock
lus17200_defconfig	- Build for lus17200
mainstone_defconfig	- Build for mainstone
mxlads_defconfig	- Build for mxlads
neponset_defconfig	- Build for neponset
netwinder_defconfig	- Build for netwinder
omap_h2_1610_defconfig	- Build for omap_h2_1610
pleb_defconfig	- Build for pleb
poodle_defconfig	- Build for poodle
pxa255-idp_defconfig	- Build for pxa255-idp
rpc_defconfig	- Build for rpc
s3c2410_defconfig	- Build for s3c2410
shannon_defconfig	- Build for shannon
shark_defconfig	- Build for shark
simpad_defconfig	- Build for simpad
smdk2410_defconfig	- Build for smdk2410
spitz_defconfig	- Build for spitz
versatile_defconfig	- Build for versatile

```

make V=0|1 [targets] 0 => quiet build (default), 1 => verbose build
make O=dir [targets] Locate all output files in "dir", including .config
make C=1 [targets] Check all c source with $CHECK (sparse)
make C=2 [targets] Force check of all c source with $CHECK (sparse)

```

Execute "make" or "make all" to build all targets marked with [*]
 For further info see the ./README file

读者可能很少使用上面列出的各种目标，甚至从来也不会使用，但是，知道有这些目标存在也是非常有用的。从代码清单4-7中可以看到，那些用星号标识出来的目标是编译的默认目标。注意那些众多的默认配置，也就是*_defconfig配置选项。回顾4.2.2节，在其中使用的编译指令应用了最原始的内核源码树完成编译工作，也就是那里使用了默认的目标和配置文件进行了内核编译工作。其中目标默认配置文件是ixp4xx_defconfig，在上面的ARM编译目标列表中也能够发现。其实，这是一种理解特定内核和体系结构下默认系统配置的一种非常好的方法。

4.3.4 内核配置

几乎在300多个内核子目录下都能够发现Kconfig文件（或者类似的具有扩展名的文件，如Kconfig.ext）。Kconfig驱动相应的子目录下内核编译的配置过程以实现相应的特性。Kconfig文件所包含的内容会由配置子系统解析，从而实现针对用户的一些配置工作，以及与配置参数相对应的简要说明文本。

配置工具（例如前文所述的gconf）将读取Kconfig文件，它将从arch子目录下开始读取Kconfig文件，并且利用Kconfig文件开始整个编译子系统的工作，相对应的文件入口如：

```
gconf: $(obj)/gconf
      $< arch/$(ARCH)/Kconfig
```

根据编译的目标体系结构的不同，gconf工具读取相应体系结构的Kconfig文件作为顶层配置的定义，在Kconfig文件中包含了很多类似下面的定义行：

```
source "drivers/pci/Kconfig"
```

这行代码指示配置工具读取另一个Kconfig文件。每种体系结构内核源码树中都包含了很多Kconfig文件，将这些文件组合起来，就构成了一个完整的配置文件菜单集合，可以用来完成内核系统的配置工作。系统源码树所包含的每个Kconfig文件都是自由定制的，利用配置工具gconf可以迭代地读取Kconfig文件并且实现最终的配置结构。

在代码清单4-8中列出了Kconfig文件的部分内容。这个Kconfig文件针对ARM体系结构，并且来自于最新的Linux 2.6版本的内核源码树。整个内核的配置由170个独立的Kconfig文件构成。受篇幅所限，这里忽略了大部分配置文件的内容，并且为了能够让大家清楚地了解问题，仅仅列出Kconfig文件的基本结构。如果把Kconfig文件所有的内容都罗列在这里，需要占据数页纸。

代码清单4-8 ARM体系结构Kconfig文件的片段

```
arch/arm/Kconfig <<<<<< (top level Kconfig)
|-> init/Kconfig
|   ...
|-> arch/arm/mach-iop3xx/Kconfig
|-> arch/arm/mach-ixp4xx/Kconfig
|   ...
|-> net/Kconfig
|   |--> net/ipv4/Kconfig
|       |--> net/ipv4/ipvs/Kconfig
|   ...
|-> drivers/char/Kconfig
|   |--> drivers/serial/Kconfig
|   ...
|-> drivers/usb/Kconfig
|   |--> drivers/usb/core/Kconfig
|   |--> drivers/usb/host/Kconfig
|   ...
|-> lib/Kconfig
```

仔细查看代码清单4-8的内容，文件arc/arm/Kconfig包含了下面一行：

```
source "net/Kconfig"
```

而在文件net/Kconfig中又包含着下面一行：

```
source "net/ipv4/Kconfig"
```

依此类推，就能够在Kconfig文件中找到更丰富的内容。

如前所述，这些Kconfig文件组合起来决定了整个系统的配置菜单结构以及相应的选项，这些结构以及选项将决定整个操作系统内核的配置。图4-3演示了配置工具（gconf）对ARM体系结构进行配置的过程，这也是代码清单4-8所定义的内核编译示例。

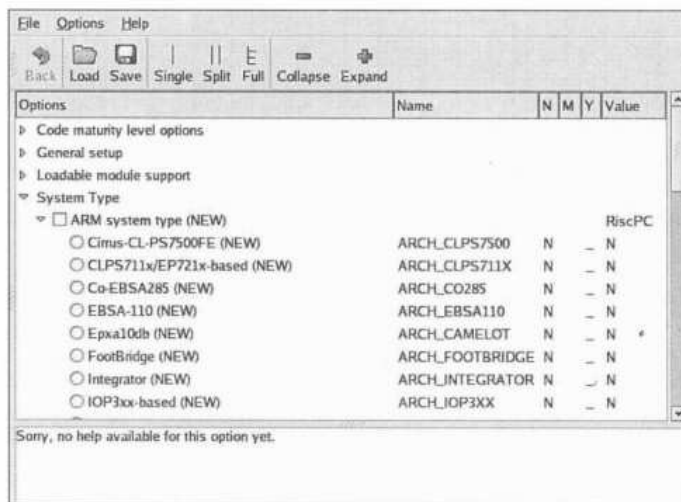


图4-3 gconf配置工具截图

4.3.5 自定义配置选项

很多嵌入式系统的开发人员都要对现有的Linux内核进行改造，以便增加必要的特性使其支持特定的自定义硬件系统。其中，最常见的应用就是针对给定的系统硬件平台多版本化而实现的多版本软件支持，每种版本都需要一些特定针对内核源码树进行配置的编译选项。当然，可以针对每个版本的硬件平台使用相对独立的内核源码树，但是更便捷的方法是增加必要的配置选项，从而实现用户自定义的特性。

前面介绍的配置管理体系结构能够简化自定义应用并且增加必要的特性。对Kconfig文件的快速讲解让大家基本了解了文件的体系结构以及配置脚本的语法。作为示例，这里假设有两个是基于IXP425网络处理器的硬件平台，开发团队把它们分别命名为Vega和Constellation，每个硬件板上都有一些特殊的硬件需要在操作系统启动的时候进行相应的初始化工作。那么通过增加必要的配置选项就能够非常容易地解决针对不同平台的配置问题。代码清单4-9列出了ARM体系结构下的Kconfig文件的片段。

代码清单4-9 .../arch/arm/Kconfig文件片段

```
source "init/Kconfig"

menu "System Type"

choice
    prompt "ARM system type"
    default ARCH_RPC

config ARCH_CLPS7500
    bool "Cirrus-CL-PS7500FE"

config ARCH_CLPS711X
    bool "CLPS711x/EP721x-based"

...

source "arch/arm/mach-ixp4xx/Kconfig"
```

从上面ARM体系结构顶层下的Kconfig文件片段中可以发现,其中定义了一个菜单项System Type。在ARM System type提示后面,可以发现一系列针对ARM体系结构的配置选项。在文件的后面,还可以找到IXP4xx系列处理器特定的定义,那么在这里就可以增加自定义的配置选项了。在代码清单4-10中就是增加了自定义属性的配置文件。同样,为了便于理解,这里忽略了一些不必要的文本内容,使用省略号替代相应的内容。

代码清单4-10 arch/arm/mach-ixp4xx/Kconfig文件内容片段

```
menu "Intel IXP4xx Implementation Options"

comment "IXP4xx Platforms"

config ARCH_AVILA
    bool "Avila"
    help
        Say 'Y' here if you want your kernel to support...

config ARCH_ADI_COYOTE
    bool "Coyote"
    help
        Say 'Y' here if you want your kernel to support
        the ADI Engineering Coyote...

# (These are our new custom options)
config ARCH_VEGA
    bool "Vega"
    help
        Select this option for "Vega" hardware support
```



```

config ARCH_CONSTELLATION
    bool "Constellation"
    help
        Select this option for "Constellation"
        hardware support

```

图4-4所示的是运行gconf配置工具（通过make ARCH=arm gconfig启动该工具）并且修改相应的配置选项时的外观，这里仅仅进行较为简单的修改，在相应的配置工具中仅仅针对两种新的硬件平台增加必要的属性^①。简而言之，在这里将看到如何使用源码树中的配置信息选择相应的对象支持新的硬件板卡。

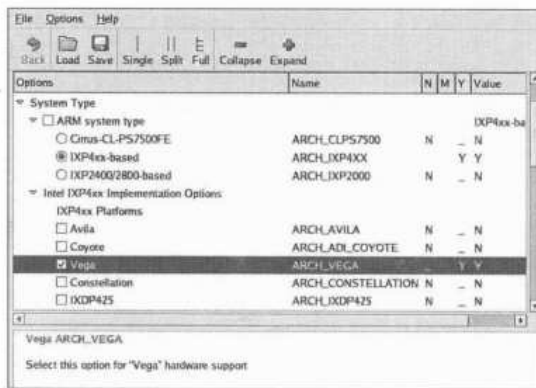


图4-4 自定义配置选项

当启动了配置文件编辑器之后（这里是gconf），就可以选择需要支持的硬件板卡之一，在.config文件中包含了具体的宏定义可以用于支持新的特性。对于所有的配置选项，每个选项都有前缀CONFIG_，用于表示内核配置选项。这里需要增加两个新的配置选项，并记录在对应的.config文件中。代码清单4-11中给出了.config文件的片段，里面包含了新的配置选项。

代码清单4-11 自定义.config文件片段

```

...
#
# IXP4xx Platforms
#
# CONFIG_ARCH_AVILA is not set
# CONFIG_ARCH_ADI_COYOTE is not set
CONFIG_ARCH_VEGA=y
# CONFIG_ARCH_CONSTELLATION is not set
# CONFIG_ARCH_IXDP425 is not set
# CONFIG_ARCH_PRPMC1100 is not set
...

```

注意，这里定义的两个新的配置选项分别针对Vega平台和Constellation平台，正如图4-4所示，

^① 在这里我们去掉了一些针对ARM体系和Intel IXP4xx处理器的选项，以适应篇幅。

在编辑工具内设置了当前支持的系统是Vega平台，在.config文件中可以发现新的CONFIG_选项表明选择了Vega硬件平台，选项值设置为'y'。同样可以看到于Constellation平台相对应的选项，只不过这里没有选择。

4.3.6 内核 makefile

当进行操作系统内核编译的时候，makefiles将扫描对应的配置属性，并且决定哪些子目录需要被引入以及哪些源代码文件在对应的配置下需要被编译。为了完成前面的示例，也就是增加了两个自定义硬件平台——Vega和Constellation，需要完成相应的makefile，这个makefile将读取配置属性并且会根据定义执行相应的动作。

在本例中需要处理相应硬件选项，假设自定义的硬件特性分别通过两个硬件设置源文件vaga_setup.c和Constellation_setup.c表示。将这两个C语言源文件保存在.../arch/arm/mach-ixp4xx路径下。代码清单4-12中给出了当前Linux发行版下完整的makefile。

代码清单4-12 .../arch/arm/mach-ixp4xx内核下的makefile

```
#
# Makefile for the linux kernel.
#

obj-y      += common.o common-pci.o

obj-$(CONFIG_ARCH_IXDP4XX)    += ixdp425-pci.o ixdp425-setup.o
obj-$(CONFIG_MACH_IXDPG425)  += ixdp425-pci.o coyote-setup.o
obj-$(CONFIG_ARCH_ADI_COYOTE) += coyote-pci.o coyote-setup.o
obj-$(CONFIG_MACH_GTWX5715)  += gtwx5715-pci.o gtwx5715-setup.o
```

读者可能奇怪这个makefile为何如此简单，因为很多工作已经在内核系统的开发过程中完成了。对于大多数开发人员，特别是只需要增加对自定义硬件支持的开发人员而言，内核构建系统的自定义化设计使得这一切都非常直接^①。

再仔细看看这个makefile，就能明显发现为了增加新的硬件特性配置需要进行的工作，只要很简单地将下面的两行代码增加到makefile中就能够实现：

```
obj-$(CONFIG_ARCH_VEGA)      += vega_setup.o
obj-$(CONFIG_ARCH_CONSTELLATION) += constellation_setup.o
```

这些步骤都完成之后就完成了本章假设的增加自定义硬件的示例。按照这些步骤，大家应该能够对自己的内核配置/编译系统进行修改以便增加对自己硬件的支持。

4.3.7 内核文档

Linux操作系统源码树本身包含了丰富的信息，不过完整读取这些文档是一件非常困难的事

^① 实际上，内核构建系统非常复杂，但是对于一般的开发人员而言，这些所谓的复杂性都被隐藏了起来，结果对于内核构建系统的修改、增删特性等操作变得非常简单，根本不需要成为专家就可以做到。

情，因为内核系统中一共有650个不同的文档分别保存在42个不同的子目录下，这些子目录都位于.../Documentation路径下。阅读这些文档时需要小心，由于Linux内核开发以及发布非常迅速，这些文档也会迅速过时。尽管如此，这些文档是学习和了解内核子系统及基本概念的很好的出发点，学习这些文档是建立相应概念的基础。

不要忽略Linux Document Project，在www.tldp.org网站上能够找到相应的信息，这些文档都能够及时得到更新^①。在本章的参考资源中提供了Linux Document Project的超链接。对于前文所述的kbuild文档在.../Documentation/kbuild路径下就能够找到。

在讨论内核文档时不涉及Google则不可能完全说明问题，很快在Merriam Webster大辞典中Googling就会变成一个新的动词！大多数情况下，很多问题已经被其他人问过并且得到了相应的答案。希望读者能够稍微花费一些时间精通一下因特网中的搜索技巧，在其上可以找到很多邮件列表或者各种类型的知识库信息，这些信息对项目开发和问题解决会非常有益。在附录D中能够找到很多有用的开源资源列表。

4.4 获取 Linux 内核

通常情况下，可以通过三种途径获取符合嵌入式系统项目需求的内核：第一种，购买商业化的嵌入式Linux内核的发行版；第二种，如果在免费使用的嵌入式Linux内核中能够找到符合项目硬件体系结构、处理器需求的内核，可以免费下载；第三种，找到与自己项目硬件体系结构、处理器要求最接近的Linux内核，下载并移植。第16章将详细介绍相关的知识。

虽然将开源的Linux内核移植到定制的系统硬件不太困难，但是这么做需要具有足够的投入，无论是资金上的投入还是工程开发人员上的投入都是必要的。这种方法可以充分使用免费的软件资源，但是进行操作系统的移植改造不一定是免费的，正如第1章所讨论的，即便是进行一个最简单的嵌入式系统应用开发，除了操作系统内核以外，还需要用到很多工具和软件组件。

还要再做些什么

本章的内容关注于Linux内核的组织结构，可能读者已经发觉，Linux内核仅仅是使用Linux操作系统的嵌入式系统中的一小部分而已。除了操作系统的内核以外，还需要下列工具或者软件才能够实现完整系统的开发、测试以及发布工作：

- 根据系统硬件平台的配置，开发或者移植系统的引导装入程序；
- 交叉编译环境以及相应的工具集，不同的硬件体系结构需要不同的工具；
- 文件系统，要针对项目所选择的硬件体系结构/处理器选择很多软件包、预先编译好的开发库等；
- 设备驱动程序，用于驱动系统中的特殊设备；
- 开发环境，主要是宿主开发工具、应用软件等；
- Linux内核源码树，针对特殊的处理器和开发板。

^① 请注意，特性的更新永远快于对应的文档开发，所以把这些文档当作相应的指导而不是实际的事实会更好一些。

所有这些组件组合起来，才构成了完整的嵌入式Linux操作系统的发行版产品。

4.5 小结

- Linux操作系统已经存在于世十多年了，目前已经是一款主流的操作系统，得到了众多的硬件体系结构的支持。
- Linux内核的官方网站是www.kernel.org，基本上所有Linux内核的发行版都能够在该网站上找到，甚至Linux 1.0版本的内核也能找到。
- 在本书中没有讨论有关Linux操作系统内核原理或者基本操作方面的内容，因为有很多书籍已经讨论得很充分了，本章的重点是内核如何组织，如何构成操作系统映像。把操作系统内核划分成易于理解的小片段是进行大规模软件项目的一种好方法。
- 本章介绍了内核系统的编译过程并且介绍了如何进行编译过程的配置。
- 目前可以选择好几种内核配置编辑器，在本章中选择一个并且通过实例介绍了如何驱动，如何修改菜单以及菜单项。这些概念对于所有的图形前端都适用。
- 内核本身包含完整的目录体系结构，包含了丰富的文档，这些有用的资源对于理解内核、学习内核及其操作有非常大的好处。
- 本章还介绍了一些获取嵌入式Linux操作系统以及发布Linux操作系统的一些知识。

参考资源

Linux 内核 HOWTO:

www.tldp.org/HOWTO/Kernel-HOWTO

内核构建文档:

<http://sourceforge.net/projects/kbuild/>

Linux文档工程:

www.tldp.org/

工具接口标准 (TIS): 可执行链接格式 (ELF) 规范, 1.2版
TIS 委员会, 1995年5月

Linux内核源码树:

.../Documentation/kbuild/makefiles.txt

Linux内核源码树:

.../Documentation/kbuild/kconfig-language.txt

Linux Kernel Development, 2nd Edition

Robert Love

Novell 出版社, 2005



本章内容

- 合成内核映像：piggy及其他
- 初始化控制流
- 内核命令行处理
- 子系统初始化
- init线程
- 小结

当嵌入式Linux系统加电后，就会有一系列复杂的事件依次发生。加电几秒钟之后，内核开始工作并且执行由系统初始化脚本中指定的一系列应用程序。这些事件的一个显著特点是，它们服从于系统的配置并且由嵌入式开发者所控制。

本章将探讨Linux内核的初始化过程，仔细研究在内核初始化中所采用的机制和处理过程，以及在初始化过程中，Linux的内核命令行信息及其在启动的时候定制Linux开发环境的用途。具备了这些知识之后，你就可以定制和控制系统初始化流程，以满足你的嵌入式系统的特殊需求。

5.1 合成内核映像：piggy 及其他

在系统加电后，嵌入式系统的引导装入程序首先取得了对处理器的控制权。在引导装入程序执行了一些基本的硬件初始化之后，控制权就会交给Linux内核。为了便于开发，这样的过程是可以手动进行的（例如在引导装入程序的提示下用户输入交互式的load/boot命令），也可以是一个自动的启动过程。我们会在第7章中讨论这个话题，所以对于引导装入程序的详细介绍将放到那里。

在第4章中，我们研究了Linux内核映像文件的组成，还记得那些用于构建体系结构的文件吗？其中一个二进制ELF格式的vmlinux文件，vmlinux文件就是内核，或者也可以称为严格意义上的内核。实际上，当在其链接过程检查vmlinux文件时，我们看到的第一行代码，在绝大多数的体系结构当中，位于一个汇编代码的源文件中，该文件称为head.S（或类似名字的文件）。在支持PowerPC系列处理器的Linux内核中，提供了head.S的几个版本，这些文件依赖于处理器。例如，AMC440系列处理器通过名为head_44x.S的文件进行初始化。

一些体系结构和引导装入程序可以直接引导vmlinux内核映像。例如，基于PowerPC体系结构和U-Boot的平台通常都可以直接引导vmlinux内核映像^①（在经过由ELF到二进制格式的转换之后，不久你就会看到）。在由其他体系结构和引导装入程序所构成的系统中，可能需要配置合适的上下文并提供必要的工具才能加载和引导内核。

代码清单5-1详细列出了编译内核过程中最后的步骤，该编译过程基于ADI Coyote参考硬件平台，该平台内含一个Intel IXP425网络处理器。该代码采用了内核编译时默认的输出格式，就像在第4章中指出的那样，这是一个有用的速记法，可以更多关注于编译过程中的错误和警告信息。

代码清单5-1 内核编译最后的流程：基于ARM/IXP425（Coyote）

```
$ make ARCH=arm CROSS_COMPILE=xscale_be- zImage
... < many build steps omitted for clarity>
LD      vmlinux
SYSMAP   System.map
OBJCOPY  arch/arm/boot/Image
Kernel:  arch/arm/boot/Image is ready
AS       arch/arm/boot/compressed/head.o
GZIP     arch/arm/boot/compressed/piggy.gz
AS       arch/arm/boot/compressed/piggy.o
CC       arch/arm/boot/compressed/misc.o
AS       arch/arm/boot/compressed/head-xscale.o
AS       arch/arm/boot/compressed/big-endian.o
LD       arch/arm/boot/compressed/vmlinux
OBJCOPY  arch/arm/boot/zImage
Kernel:  arch/arm/boot/zImage is ready
Building modules, stage 2.
...
```

在代码清单5-1中的第3行可以看到，vmlinux内核映像（严格意义上的内核）在这里被链接，之后，大量附加的目标模块得以处理，这些目标模块包括head.o、piggy.o^②以及和特定体系结构有关的head-xscale.o。在这些处理过程中，每一行的处理中都使用了相应的标识。例如，其中的AS表示汇编程序被调用，GZIP表示的是压缩等。通常来说，这些目标模块是与给定的体系结构（本例当中的体系结构是ARM/Xscale）有关的，并且包含该特定体系结构下引导内核所需的基本程序。表5-1详细列出了这些内容的组成部分。

表5-1 基于ARM/Xscale体系结构的基本目标文件

组 件	功能/描述
vmlinux	严格意义上的内核，采用ELF格式，包括符号、注释、调试信息（如果采用-g选项编译）和通用体系结构组件
System.map	描述vmlinux模块的内核符号表，基于文本格式

- ① 内核映像文件几乎总是压缩后存储的，除非启动时间是关键因素。在这种情况下，内核映像可以称为uImage，是带U-Boot首部的经压缩的vmlinux文件。参见第7章。
- ② piggy最初用来描述vmlinux映像文件的构成内容，这里的意思是，二进制的内核映像与第二阶段引导装入程序（bootstrap loader）相结合来产生复合的内核映像。

(续)

组 件	功能/描述
Image	二进制内核模块, 去掉了符号、标记、注释
head.o	针对ARM类处理器的启动代码, 就是通过这个目标文件, 引导装入程序取得了控制权
piggy.gz	采用gzip压缩的Image文件
piggy.o	piggy.gz文件的汇编格式, 可以被后面的misc.o文件所链接
misc.o	用于解压缩内核映像 (piggy.gz) 的程序, 大家所熟悉的在某些体系结构上的启动信息“Uncompressing Linux ... Done”就来源于该文件
head-xscale.o	XScale系列处理器的初始化目标文件
big-endian.o	一个小的汇编程序, 可以将XScale处理器转换为对大端字节序模式 (big-endian mode) 的支持
vmlinux	合成内核映像, 注意这是一个不合适的命名, 因为它与实际意义上的内核同名, 二者并不相同。严格意义上的内核链接了该表中的目标文件后生成该合成映像文件, 参见相关解释内容
zImage	最终的合成映像文件, 可以被引导装入程序引导, 在下文中会具体介绍

图5-1有助于你理解该结构以及后续的讨论内容, 它展示了内核映像文件的组成部分, 以及产生一个最终可引导内核映像文件的过程, 下面章节会详细介绍这些组成部分和映像文件的产生过程。

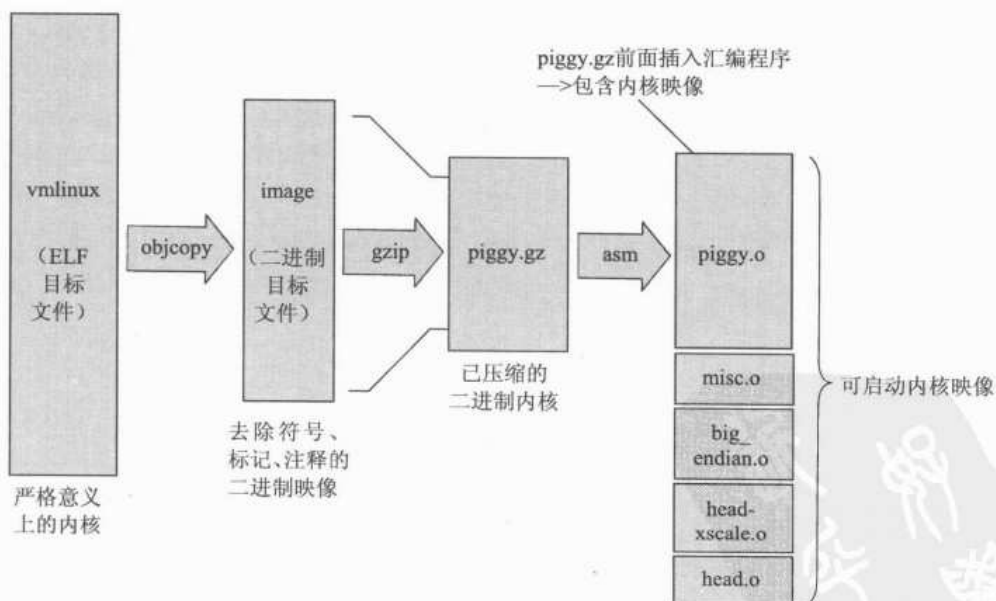


图5-1 合成内核映像文件的结构

5.1.1 Image 目标文件

在编译好ELF格式的vmlinux内核文件之后, 内核将会继续处理在表5-1中描述的目标文件,

Image目标文件是由vmlinux目标文件所创建的。Image文件基本上是去除了多余段（标记和注释）的vmlinux ELF文件，同时也去掉了可能已经存在的调试符号。创建Image文件的命令如下：

```
xscale_be-objcopy -O binary -R .note -R .comment -S \
vmlinux arch/arm/boot/Image
```

命令中的-O选项指示objcopy工具产生一个二进制文件，-R选项的作用是去掉名为.note和.comment的ELF段，而-S选项是去掉调试符号的标记。可以看到objcopy工具以ELF格式的vmlinux作为输入文件产生了二进制目标文件Image。概括地讲，Image就是二进制格式的内核文件，它去掉了调试符号、.note和.comment ELF段。

5.1.2 体系结构相关的目标文件

随着过程的深入，许多小的模块被编译进来，包括一些执行底层体系结构和处理器任务的汇编程序（如head.o、head-xscale.o等）。每一个这样的目标文件均在表5-1中有简要介绍，特别需要注意的是，这个过程会产生一个名为piggy.o的目标文件。首先，采用如下的命令压缩Image文件（二进制内核映像文件）：

```
gzip -f -9 < Image > piggy.gz
```

执行该命令会生成一个名为piggy.gz的压缩文件，它只是二进制内核文件Image的压缩版本，你可以在图5-1中看到图形化的压缩过程。接下来的内容更加有趣，一个名为piggy.S的汇编文件被汇编，该文件包括对压缩文件piggy.gz的说明。在本质上，二进制内核映像Image要依附于一个执行底层硬件初始化任务的工具——第二阶段引导装入程序（bootstrap loader）^①，第二阶段引导装入程序由一些汇编程序构成。该第二阶段引导装入程序会对处理器和必要的内存区域进行初始化，解压缩二进制内核映像并且在转交对系统的控制权之前把它加载到系统内存的合适位置。代码清单5-2当中详细列出了.../arch/arm/boot/compressed/piggy.S中的内容。

代码清单5-2 汇编文件piggy.S

```
.section .piggydata,#alloc
.globl input_data
input_data:
.incbn "arch/arm/boot/compressed/piggy.gz"
.globl input_data_end
input_data_end:
```

这个小的汇编文件（piggy.S）虽然简单，但是却产生了一个复杂而又不容易看到的结果。使用这个汇编文件的目的是利用名为.piggydata的汇编ELF段来解压压缩的二进制内核映像，该过程由.incbn汇编文件预处理调用，它可以视为一个#include文件的汇编版。简而言之，汇编文件piggy.S的作用是将包含压缩的二进制内核映像文件作为另一个映像文件——第二阶段引导装入程序的有效内容。注意piggy.S代码中的input_data和input_data_end标签，第二阶

① 不要与引导装入程序（bootloader）相混淆，第二阶段引导装入程序（bootstrap loader）可视为第二阶段的引导装入程序，而引导装入程序可视为第一阶段的引导装入程序。

段引导装入程序利用它们来识别该有效内容在内核映像中的范围。

5.1.3 第二阶段引导装入程序

不要与引导装入程序混淆,许多体系结构采用第二阶段引导装入程序将Linux内核映像文件加载到内存中。一些第二阶段引导装入程序会对内核映像进行校验,而绝大多数会对内核映像进行解压缩并且对其进行重新部署。引导装入程序和第二阶段引导装入程序的区别也很简单,引导装入程序在开发板加电之后取得控制权,而且它在任何情况下都不会依赖内核;而第二阶段引导装入程序的主要作用是充当开发板级的引导装入程序和内核之间的纽带。正是第二阶段引导装入程序为内核运行提供了合适的上下文,同时也执行了一些必要的操作,如解压并且重新部署二进制内核映像。它的作用类似于PC体系结构的第一和第二阶段的装入程序。

图5-2会使这个概念更加清晰,第二阶段引导装入程序与内核映像联系在一起用于系统引导。



图5-2 基于ARM Xscale 平台的合成内核映像文件

在我们研究的这个例子中,组成第二阶段引导装入程序的二进制映像如图5-2所示。它具有如下功能:

- 处理器底层的初始化,包括支持处理器内部指令及数据缓存,关闭系统中断,建立C运行环境。这些功能包括在head.o和head-xscale.o中。
- 解压缩和重定位代码,包含在misc.o中。
- 其他特定于处理器的初始化,例如big-endian.o文件,实现对一些特殊处理器大端字节序模式的支持。

前面章节中对基于ARM/XScale平台下内核执行细节的研究值得关注,每一个体系结构有不同的细节,尽管这些内容是类似的。在这里给出一个类似的分析过程,你就会掌握自身所用的体系结构的需求。

5.1.4 引导信息

你或许已看过一台PC工作站上的Linux引导过程,如Red Hat或SUSE Linux。在PC启动BIOS

信息之后，用户会看到Linux在控制台上的一些急促的输出信息，这些信息反映的是对不同内核子系统的初始化。对于不同体系结构的平台来说，输出内容中的重要部分是共同的。输出信息中前两条简短有趣的引导信息是关于内核的版本号和内核命令行，接下来的内容会对它们进行详细的介绍。代码清单5-3显示了ADI采用了Intel XScale IXP425处理器的Coyote开发平台下引导Linux的信息，为了便于阅读已经对其中的代码格式进行了整理。

代码清单5-3 IPX425的Linux引导信息

```

1 Uncompressing Linux... done, booting the kernel.
2 Linux version 2.6.14-clh (chris@pluto) (gcc version 3.4.3 (MontaVista 3.4.3-
↳ 25.0.30 .0501131 2005-07-23)) #11 Sat Mar 25 11:16:33 EST 2006
3 CPU: XScale-IXP42x Family [690541c1] revision 1 (ARMv5TE)
4 Machine: ADI Engineering Coyote
5 Memory policy: ECC disabled, Data cache writeback
6 CPU0: D VIVT undefined 5 cache
7 CPU0: I cache: 32768 bytes, associativity 32, 32 byte lines, 32 sets
8 CPU0: D cache: 32768 bytes, associativity 32, 32 byte lines, 32 sets
9 Built 1 zonelists
10 Kernel command line: console=ttyS0,115200 ip=bootp root=/dev/nfs
11 PID hash table entries: 512 (order: 9, 8192 bytes)
12 Console: colour dummy device 80x30
13 Dentry cache hash table entries: 16384 (order: 4, 65536 bytes)
14 Inode-cache hash table entries: 8192 (order: 3, 32768 bytes)
15 Memory: 64MB = 64MB total
16 Memory: 62592KB available (1727K code, 339K data, 112K init)
17 Mount-cache hash table entries: 512
18 CPU: Testing write buffer coherency: ok
19 softlockup thread 0 started up.
20 NET: Registered protocol family 16
21 PCI: IXP4xx is host
22 PCI: IXP4xx Using direct access for memory space
23 PCI: bus0: Fast back to back transfers enabled
24 dmabounce: registered device 0000:00:0f.0 on pci bus
25 NetWinder Floating Point Emulator V0.97 (double precision)
26 JFFS2 version 2.2. (NAND) (C) 2001-2003 Red Hat, Inc.
27 Serial: 8250/16550 driver $Revision: 1.90 $ 2 ports, IRQ sharing disabled
28 ttyS0 at MMIO 0xc8001000 (irq = 13) is a XScale
29 io scheduler noop registered
30 io scheduler anticipatory registered
31 io scheduler deadline registered
32 io scheduler cfq registered
33 RAMDISK driver initialized: 16 RAM disks of 8192K size 1024 blocksize
34 loop: loaded (max 8 devices)
35 eeepro100.c:v1.09j-t 9/29/99 Donald Becker
↳ http://www.scyld.com/network/eeepro100.html
36 eeepro100.c: $Revision: 1.36 $ 2000/11/17 Modified by Andrey V. Savochkin
↳ <saw@saw.sw.com.sg> and others
37 eth0: 0000:00:0f.0, 00:0E:0C:00:82:F8, IRQ 28.
38 Board assembly 741462-016, Physical connectors present: RJ45
39 Primary interface chip i82555 PHY #1.
40 General self-test: passed.
41 Serial sub-system self-test: passed.
```

```

42 Internal registers self-test: passed.
43 ROM checksum self-test: passed (0x8b51f404).
44 IXP4XX-Flash.0: Found 1 x16 devices at 0x0 in 16-bit bank
45 Intel/Sharp Extended Query Table at 0x0031
46 Using buffer write method
47 cfi_cmdset_0001: Erase suspend on write enabled
48 Searching for RedBoot partition table in IXP4XX-Flash.0 at offset 0xfe0000
49 5 RedBoot partitions found on MTD device IXP4XX-Flash.0
50 Creating 5 MTD partitions on "IXP4XX-Flash.0":
51 0x00000000-0x00060000 : "RedBoot"
52 0x00100000-0x00260000 : "MyKernel"
53 0x00300000-0x00900000 : "RootFS"
54 0x00fc0000-0x00fc1000 : "RedBoot config"
55 mtd: partition "RedBoot config" doesn't end on an erase block -- force
read-only 0x00fe0000-0x01000000 : "FIS directory"
56 NET: Registered protocol family 2
57 IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
58 TCP established hash table entries: 4096 (order: 2, 16384 bytes)
59 TCP bind hash table entries: 4096 (order: 2, 16384 bytes)
60 TCP: Hash tables configured (established 4096 bind 4096)
61 TCP reno registered
62 TCP bic registered
63 NET: Registered protocol family 1
64 Sending BOOTP requests . OK
65 IP-Config: Got BOOTP answer from 192.168.1.10, my address is 192.168.1.141
66 IP-Config: Complete:
67     device=eth0, addr=192.168.1.141, mask=255.255.255.0, gw=255.255.25
↳ 5.255,
68     host=192.168.1.141, domain=, nis-domain=(none),
69     bootserver=192.168.1.10, rootserver=192.168.1.10,
↳ rootpath=/home/chris/sandbox/coyote-target
70 Looking up port of RPC 100003/2 on 192.168.1.10
71 Looking up port of RPC 100005/1 on 192.168.1.10
72 VFS: Mounted root (nfs filesystem).
73 Freeing init memory: 112K
74 Mounting proc
75 Starting system loggers
76 Configuring lo
77 Starting inetd
78 / #

```

就像在代码清单5-3中看到的那样, Linux在内核的启动过程中输出了许多有用信息, 我们将在后面章节中详细研究这些输出信息的内容。代码清单5-3中的第一行信息就是由前面提到的第二阶段引导装入程序所产生的, 该条由解压缩引导代码所产生的信息可在.../arch/arm/boot/compressed/misc.c中看到。

代码清单5-3中第二行是关于内核的版本信息, 它是内核输出的第一条信息, 内核start_kernel()函数是用于产生头几行启动信息所执行的C代码(在.../init/main.c), 如下所示:

```
printk(linux_banner);
```

该语句会输出代码清单5-3中第二行那样的内核版本描述信息, 该信息包括了如下一些与内核映像相关的内容。

- 内核版本: 2.6.10-clh;
- 编译内核的用户/机器名;
- 编译工具链信息: gcc version 3.4.3, 由MontaVista Software提供;
- 构建号 (build number);
- 编译的日期时间。

上面提到的这些信息对于开发过程和后续的投产都是十分有用的, 上面的几行除了构建号一项, 其他都很好理解。构建号是开发者在对内核修改时使用的一种简单工具, 通过构建号用户可以在版本信息中加入比时间、日期更真实的信息, 这是开发者使用的一种常用的自动加入构建记录的方法, 你在本例中可以看到对应的信息是该系列的第11次构建, 如代码清单5-3中的第二行所示。版本信息存放在顶层目录下名为`.version`的隐藏文件中, 它会由`.../scripts/mkversion`下的构建脚本和顶层目录下的`makefile`文件自动更新。简单来说, 它是当内核在任何地方被重新构建之后自动更新的版本信息。

5.2 初始化控制流

我们对合成内核映像的结构和组成内容已经有了一定的理解, 现在来研究从引导装入程序到内核的整个引导过程。如第2章所述, 引导装入程序是驻留在非易失性存储器 (Flash或ROM) 中的底层组件, 它会在系统加电后即取得对系统的控制权。引导装入程序是一个精简的程序, 主要用来完成底层的初始化任务、加载内核映像和系统的自我诊断, 它包含内存转储 (memory dump) 和内存填充 (memory fill) 程序以修改和检查内存中的内容, 同时也包含了底层的自检程序, 包括对内存和I/O口的检测。最后, 引导装入程序包含了一个用于引导并且把对系统的控制权交给另一个程序的逻辑, 这个程序通常就是操作系统, 例如Linux。

本章以ARM/XScale平台为例, 该平台所使用的引导装入程序版本为Redboot, 当系统加电之后, 引导装入程序即被调用并且开始加载操作系统 (OS)。当引导装入程序定位到操作系统映像并且引导操作系统映像 (存在于本地Flash、硬盘设备、本地局域网或其他设备) 之后, 引导装入程序就将系统的控制权交给了该映像。

在这个特定XScale平台中, 引导装入程序将控制权交给第二阶段引导装入程序中标签为`start`的`head.o`模块, 这一过程如图5-3所示。

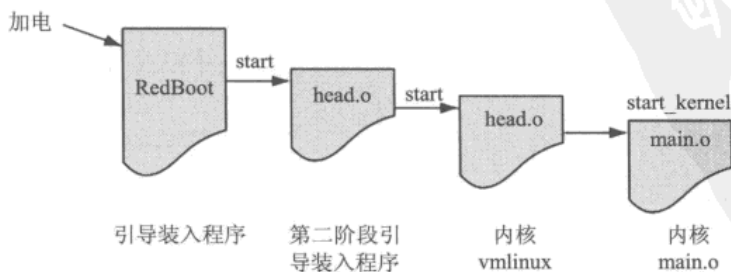


图5-3 ARM引导流程控制

前面已详细讨论了, 第二阶段引导装入程序会预先认为, 内核映像会创建一个适当的环境以对内核进行解压缩和重定位, 并且将对系统的控制权传递给内核映像。对于大多数体系结构来说, 对系统的控制权会由第二阶段引导装入程序直接传递给严格意义上的内核, 即传递给名为head.o的模块。遗憾的是, 由于历史和人为的原因, 第二阶段引导装入程序和严格意义上的内核都包括一个head.o模块, 这对于一些嵌入式Linux开发的入门者来说就造成了混淆。我怀疑用户最终能否接受, 觉得将第二阶段引导装入程序中的head.o模块称为kernel_bootstrap_loader_head.o似乎更合适一些。事实上, 最新的Linux 2.6源码树中包含了不少于37个都命名为head.s的源文件, 这也就是为什么你需要了解内核源码树的另一个原因。

回到图5-3中引导过程的图形化界面。当第二阶段引导装入程序完成任务之后, 严格意义上的内核head.o就取得了对系统的控制权, 并且从main.c程序中的start_kernel()函数开始执行。

5

5.2.1 内核入口点: head.o

内核开发者都希望这个与处理器体系结构相关的head.o模块变得非常通用, 尽量减少对特定处理器(machine)^①的依赖。该模块由汇编文件head.S而来, 所在路径为.../arch/<ARCH>/kernel/head.S。具体到实际应用, <ARCH>用给定的处理器体系结构来代替。本章基于ARM/XScale平台的例子中, 你可以看到这里的<ARCH>是arm。

head.o模块需要针对内核主体进行体系结构级或经常是CPU级的初始化。CPU级的初始化在同系列处理器家族中尽可能地保持了通用性, 而体系结构级的初始化在其他处执行, 不久你就会看到。在这些低级别的初始化任务中, head.o模块所执行的功能如下所示:

- 检测处理器及整个体系结构的合法性;
- 创建初始化页表目录;
- 支持处理器的内存管理单元(MMU);
- 进行错误检测并且生成相应报告;
- 跳转到严格意义上的内核的起始处main.c。

该模块的功能非常复杂, 许多嵌入式开发的初学者试图单步调试各个部分的代码, 但最终都失败了。尽管对于汇编语言复杂性和虚拟内存硬件细节的讨论超出了本书的范围, 但是这个复杂的模块仍有一些特点值得注意。

当第二阶段引导装入程序将控制权交给内核的head.o模块时, 处理器在过去称为实模式(real mode, 用x86体系结构的术语讲)的情况下操作。事实上, 逻辑地址包含在处理器的程序计数器^②(或与其相关的寄存器)中, 该逻辑地址实际上是由处理器的存储器地址总线引脚送出的实际物理地址。处理器的寄存器和内核的数据结构很快就会初始化从而可以支持内存的数据传输, 同时, 处理器的内存管理单元会被打开, 这将会突然导致处理器所识别的地址空间被用户指定的虚拟地址列表所取代。简单地讲, 一旦启动内存管理单元的功能, 那么实际的物理地址就会被逻

^① 这里所用的术语machine指的是一个特定的硬件参考平台。

^② 也经常叫作指令指针, 是指寄存器保存下一条在内存中机器指令的地址。

辑地址所取代，这也就是为什么调试者不能像调试普通代码那样来对该模块的各个部分进行调试的原因。

第二个需要注意的地方是，这个处于引导过程早期阶段的环节具有有限映射的特点。许多开发者在为了他们特定的开发平台^①而修改head.o时，对于这个限制感到非常迷惑。假定一个硬件设备在引导过程的前期要进行固件程序的加载，一种解决方案是将必要的固件程序静态编译到内核映像中，然后下载到用户设备。然而，由于在该阶段有限的内存映射，用户的固件程序大小就非常有可能超过在早期引导阶段对映射区域的限制，当执行用户程序时，由于用户试图访问处理器内部无效映射的内存区域，就会导致页错误的结果。更糟糕的是，对于页错误的处理工具在这个早期阶段并没有安装，所以其结果就会导致莫名其妙的系统崩溃。因此，在这个早期的引导阶段，用户几乎不会得到任何错误信息来帮助用户排查错误。

考虑延迟（在对内核引导之后）对用户系统的硬件初始化是一个明智的选择，如果可能，用户可以依靠众所周知的设备驱动方式来访问用户硬件设备，而不是试图通过定制非常复杂的汇编启动代码来实现，大量由非正式文档所记录的技术手段可以在这种方式中用到。如果用户采取了修改汇编启动代码的方式，那么就会在开发时间、费用以及开发的复杂程度上付出更高的代价。当一次很小的硬件改动可以有效地节省软件开发时间时，软件工程师和硬件工程师需要在早期的硬件开发阶段对此进行讨论。

认识到在一个虚拟内存环境中开发者所面临的某些限制非常重要，许多有经验的嵌入式开发者对于在虚拟内存环境下的开发缺乏或者几乎没有什么经验，早先提到的情况只是对于这个缺陷说明而且有待开发者在虚拟内存中使用的一个例子。当今几乎所有32位或者更高级的微处理器已经具有用来实现虚拟内存体系结构的内存管理硬件单元。采用虚拟内存技术处理器的一个最显著的优势是，在开发组成员开发大型复杂应用程序时，不受编程错误的影响，同时保护了其他软件模块以及内核本身。

5.2.2 内核启动: main.c

内核的head.o模块执行的最后一项任务是将系统控制权移交给内核的主要启动文件（采用C语言编写），我们用本章其余的内容来介绍这个重要的文件。

对于每一种不同的处理器体系结构来说，实现系统控制权移交的方法不同，但是每一种体系结构的head.o模块都会用一个类似的结构来实现将系统的控制权移交给严格意义上的内核。例如，ARM系列的处理器会使用如下比较简单的方法：

```
b    start_kernel
```

对于PowerPC，会是类似于下面这样的内容：

```
lis    r4,start_kernel@h
ori    r4,r4,start_kernel@l
lis    r3,MSR_KERNEL@h
ori    r3,r3,MSR_KERNEL@l
mtspr  SRR0,r4
```

^① 为用户平台修改head.s非常容易导致失败，这里有一个相对较好的办法，见第16章中的参考阅读内容。

```
mtspr SRR1,r3
rfi
```

这里不准备详细描述上述语句的汇编语法，这两个例子的结果是一样的。系统控制权由内核链接的第一个目标模块（head.o）交给了位于.../init/main.c中的C语言函数start_kernel()，就是从这里内核开始了它的生命历程。

如果想要对Linux内核有更深入的理解就应该仔细研究这里的main.c文件，了解其组成部分以及它们是如何被初始化的。Linux内核的所有启动任务都由main.c完成，通过初始化，内核的第一个线程将挂载一个根文件系统，同时会执行用户空间下的第一个应用程序。

start_kernel()函数是main.c程序中最庞大的函数，绝大多数Linux内核的初始化工作都是通过该函数来实现的。这里，我们着重强调了这些特殊的任务，这些初始化任务将被证明在嵌入式系统开发中是十分有用的。有必要重申的是：如果用户想要对开发Linux内核有更好的理解，那么花费时间去研究main.c中的代码将是一个非常好的方法。

5

5.2.3 体系结构设置

在.../init/main.c程序中，start_kernel()函数所要处理的第一件事情是调用setup_arch()函数，该函数为单参数，参数为指向内核命令行的指针。关于内核命令行已在前面提到，在下一节将详细介绍。

```
setup_arch(&command_line);
```

该语句调用了与体系结构有关的配置程序用来执行初始化任务。在这些初始化任务中，setup_arch()函数调用一些函数来识别给定的CPU，并且提供一种机制来调用高级别的CPU初始化程序，其中被setup_arch()直接调用的一个函数是setup_processor()，该函数所在路径是.../arch/arm/kernel/setup.c。它可以识别出CPU的ID号和版本号，同时可以调用特定CPU的初始化功能，此外还会在引导过程中通过控制台输出一些相关信息。

在代码清单5-3中可以看到这些输出信息的例子，第3~8行的信息即是。在这些信息中，你可以看到CPU类型、ID序列号和从处理器内核中直接读出的修订版本号。接下来的信息详细描述了处理器缓存的类型及大小。在代码清单5-3所示的例子中，IXP425处理器具有一个32KB的指令缓存和一个32KB的数据缓存，同时还有该处理器内部缓存的一些其他细节信息。

体系结构设置的最后工作之一是执行与硬件平台级相关的初始化工作，这与体系结构级初始化的机制是不同的。对于ARM，你会在.../arch/arm/下看到一系列的mach-*目录，这些目录与硬件平台级的初始化相关，取决于用户的硬件平台类型。MIPS体系结构也会包含一些用来支持参考平台的目录。对于PowerPC，也有一些依赖于硬件平台的结构用以支持许多通用的启动功能。我们将在第16章中详细介绍。

5.3 内核命令行处理

在启动代码main.c执行完早期的一些内核初始化任务之后，就会显示内核的命令行信息。为方便起见，这里重新列出代码清单5-3中的第10行内容：

Kernel command line: console=ttyS0,115200 ip=bootp root=/dev/nfs

在这个简单的例子中，引导中的内核在串行设备ttyS0（通常是第一个串口）上打开一个控制台，通信波特率设定为115Kbit/s。此外，它还通过一个BOOTP服务器获得自身的初始化IP地址，并且通过NFS协议挂载根文件系统。（我们将在第12章讲到BOOTP、在第9章和第12章中讲到NFS。现在我们只是讨论Linux内核的命令行机制。）

引导装入程序或第二阶段引导装入程序通过一系列被称为内核命令行的参数实现对Linux的引导。尽管在实际中并不是通过shell命令提示来调用内核，但是许多版本的引导装入程序常常采用将参数传递给Linux内核这种非常流行的模式。某些平台上的引导装入程序不能很好地识别Linux，那么就在内核编译时定义内核命令行参数，并且将其作为Linux内核二进制映像固件代码的一部分。而在另一些平台（例如运行Red Hat Linux的桌面PC）中，命令行参数可以由用户修改而不用重新编译内核。第二阶段引导装入程序（在PC中是Grub或Lilo）通过一个配置文件建立内核命令行并且在内核引导过程中传递给内核。这些命令行参数是一种引导机制，用来在给定硬件平台上设置为正确引导所需的初始化配置。

Linux在整个内核中定义了大量的命令行参数。在Linux源码中的.../Documentation子目录中有一个名为kernel-parameters.txt文件，该文件包含了Linux内核命令行参数列表，它们按字母顺序依次列出。前面提到关于内核文档的警告信息：内核的变化要快于内核文档的变化，因此，可以以该文件为向导，但它并不是一个最权威的参考。在kernel-parameters.txt文件中有超过400多个内核命令行参数，但这并不是所有的内核命令行参数，所以必须直接查阅源代码。

Linux内核命令行参数的基本语法比较简单，大部分从代码清单5-3第10行里很容易看到。内核命令行参数的形式可以是单个单词、key=value对或key= value1, value2, ...等复合形式。通过使用这些信息进行数据传递，所有的命令行都是可用的并且可以由许多的模块来处理。前面提到的main.c中的setup_arch()函数就是通过内核命令行参数调用的。通过这种调用，可以向体系结构级或硬件平台级相关代码中传递参数和配置指令。

设备驱动程序编写者和内核开发者都可以为他们特定的需要而增加相应的命令行参数。我们来看一下这种方式的实现机制。遗憾的是，在处理这些内核命令行参数的时候会涉及一些复杂的因素，首先就是原先的机制将受到抑制以便实现更为健壮的系统。第二个难点是我们需要掌握复杂的链接脚本以全面理解这种实现机制^①。

__setup 宏

可以考虑将控制台设备作为使用内核命令行参数的一个例子。我们希望该设备在内核引导的早期阶段就初始化，这样在引导过程中控制台信息就可以通过该设备输出，该初始化过程创建在名为printk.o的内核目标文件中，其C源代码位于.../kernel/printk.c。执行控制台初始化的函数是console_setup()，该函数将内核命令行的参数作为其唯一的参数。

^① 这些难点并不是必需的。事实上，大多数人不需要理解链接脚本，只有嵌入式工程师需要。在本章的最后对GNU LD参考手册有很好的说明。

配置程序和设备驱动程序与在内核命令行中所指定控制台参数进行通信的难点,在于要求该参数是标准通用的模式。该情形更复杂的情况是,命令行参数在那些模块调用它们之前(或就在此时)就要用到。在文件main.c中的启动代码里,在内核命令行进行主要处理的位置,如果没有每一个参数的使用信息,就不可能知道这几百个内核命令行参数中每一个参数的目标函数,所以需要一种灵活通用的方法将内核命令行参数传递给其使用者。

对于Linux 2.4或更早的版本,开发者通过使用一个简单的宏来解决上述问题。尽管没有得到重视,但是__setup宏仍然在整个Linux内核中得到了广泛使用。在后续内容中,我们会使用代码清单5-3中的内核命令行来演示__setup是如何工作的。

从代码清单5-3的第10行可知,下面的内容即是第一个传递给内核的完整的命令行参数:

```
console=ttyS0,115200
```

引用该例子的真正目的并不在于命令行参数的实际含义,而在于说明其工作机制,所以如果你没有理解该参数或参数值并不要紧。

代码清单5-4的内容是.../kernel/printk.c中的一部分代码,其中去掉了函数的主体部分,因为它与这里讨论的内容无关,我们关心的只是在代码清单5-4中列出的内容,即对__setup的宏调用。__setup宏在这里有两个参数:一个字符串参数和一个函数指针。传递给__setup宏的字符串与第一个与内核命令行相关的8字符的参数console=一致是绝非偶然的。

代码清单5-4 控制台设置部分代码

```
/*
 * Setup a list of consoles. Called from init/main.c
 */
static int __init console_setup(char *str)
{
    char name[sizeof(console_cmdline[0].name)];
    char*s, *options;
    int idx;

    /*
     * Decode str into name, index, options.
     */

    return 1;
}

__setup("console=", console_setup);
```

你可以将__setup宏看作是内核命令行控制台参数在内核中的注册函数。当字符串信息console=出现在内核命令行时,就通过__setup宏的第2个参数调用函数console_setup()。但是在并不知道控制台功能的情况下,这个模块之外的配置代码是如何获取该信息呢?事实上,其实现机制巧妙而复杂,并且依赖于目标链接器所创建的列表。

真正的细节隐藏于一系列的宏当中,这些宏通过在一部分目标代码中增加段属性(或其他属性)用来隐藏。目标文件会联合函数指针(function pointer)依字母顺序建立一个静态列表,该

列表会由最终vmlinux ELF映像中一个独立ELF段的编译器发出。理解上述技术细节非常重要，它在内核中许多进行特殊处理的地方都要用到。

我们来看看对于__setup宏这是如何实现的。代码清单5-5是定义了__setup宏系列的头文件.../include/linux/init.h下的部分内容。

代码清单5-5 init.h下的__setup宏的定义

```
...
#define __setup_param(str, unique_id, fn, early)          \
    static char __setup_str_##unique_id[] __initdata = str; \
    static struct obs_kernel_param __setup_##unique_id    \
        __attribute__((__section__(".init.setup")))      \
        __attribute__((aligned(sizeof(long))))           \
        = { __setup_str_##unique_id, fn, early }

#define __setup_null_param(str, unique_id)               \
    __setup_param(str, unique_id, NULL, 0)

#define __setup(str, fn)                                 \
    __setup_param(str, fn, fn, 0)

...
```

代码清单5-5是语法乏味的定义。回想代码清单5-4，我们最初所调用的__setup宏的形式如下：

```
__setup("console=", console_setup);
```

经过稍稍简化，编译器在宏扩展后，其预处理器产生如下结果：

```
static char __setup_str_console_setup[] __initdata = "console=";
static struct obs_kernel_param __setup_console_setup \
    __attribute__((__section__(".init.setup"))) = \
    { __setup_str_console_setup, console_setup, 0};
```

为了增加可读性，将上述结果的第2行和第3行采用UNIX的行续符“\”分隔开来。

我们故意略去了两个和本次讨论内容无关的编译器属性。简要地说，__attribute_used__（本身就是一个隐藏了很多语法细节的宏）会告诉编译器发出一个函数或变量，即使在编译过程中并没有用到任何优化参数^①。__attribute__((aligned))会告诉编译器按照特定的边界来对齐结构，在本例中是sizeof(long)。

简化处理后剩下的就是这种机制的核心部分。首先，编译器会产生名为__setup_str_console_setup[]的初始化后字符数组，该数组包含console=字符串信息；其次，编译器会产生一个包含三个成员的结构：指向内核命令行字符串（在字符数组中声明）的指针、指向配置函数本身的指针和一个简单的标识。这里的关键在于依附于结构的段属性，该属性会通知编译器将

^① 通常情况下，如果一个变量被定义为static类型并且在编译过程中从未引用到，那么编译器就会有警告信息，因为这些变量并没有被明确引用，编译器就会报一些警告信息。

该结构送到ELF目标模块内名为`.init.setup`的特殊段中。在这个链接阶段，所有由`__setup`宏定义的结构一起被放置到这个`.init.setup`段中，实际结果就是创建了一个包含这些结构的数组。代码清单5-6是`.../init/main.c`中的一部分内容，它们说明了这个数据是如何获取和使用的。

代码清单5-6 内核命令行处理

```

1 extern struct obs_kernel_param __setup_start[], __setup_end[];
2
3 static int __init obsolete_checksetup(char *line)
4 {
5     struct obs_kernel_param *p;
6
7     p = __setup_start;
8     do {
9         int n = strlen(p->str);
10        if (!strncmp(line, p->str, n)) {
11            if (p->early) {
12                /* Already done in parse_early_param? (Needs
13                 * exact match on param part) */
14                if (line[n] == '\0' || line[n] == '=')
15                    return 1;
16            } else if (!p->setup_func) {
17                printk(KERN_WARNING "Parameter %s is obsolete,"
18                       " ignored\n", p->str);
19                return 1;
20            } else if (p->setup_func(line + n))
21                return 1;
22        }
23        p++;
24    } while (p < __setup_end);
25    return 0;
26 }

```

对该段代码解释还算简单。函数由一个在`main.c`文件中其他地方解析的单命令行参数调用。在这个例子中，我们要讨论的指针`line`指向字符串`console=ttyS0,115200`，它是内核命令行的一个组成部分。两个外部结构指针`__setup_start`和`__setup_end`是在一个链接脚本文本文件中定义的，而不是定义在C文件或头文件中。对于`obs_kernel_param`结构数组用来标记该数组起始和结束的标签则存在于目标文件的`.init.setup`段中。

在代码清单5-6中，通过指针`p`对这个特殊的内核命令行参数寻找匹配信息的过程，对整个结构都进行了扫描。具体在本例中，代码要为字符串信息`console=`寻找匹配信息，在这个相关的结构中，函数返回一个指向`console_setup()`函数的指针，它会以该参数（字符串`ttyS0,115200`）作为其唯一的函数参数，这一处理过程会在内核命令行处理完毕之前不停地重复。

采用所描述的这种机制将目标对象存放到ELF段的列表中，这种机制在内核中的许多地方都用到了。另一个采用这种机制的例子是，使用`__init`宏系列将初始化程序放到目标文件中一个普通的段中。与其很相近的`__initdata`被`__setup`宏用来标记为只在初始化过程中用到的数据。使用这些宏标记的初始化函数和数据被集中放到ELF段中，接下来，当使用了这些用来初始化的

函数和数据之后，内核会释放之前它们所占用的内存空间。你也许在引导过程的最后阶段看到过类似的内核信息：“Freeing init memory: 296K.”。不同用户对这些函数和数据的使用可能不尽相同，但是如果有三分之一兆，就值得使用__init宏系列，这也恰恰就是使用前面声明的__setup_str_console_setup[]数组里的__initdata宏的目的所在。

你也许会对代码清单5-6中的obsolete_符号感到迷惑，这是因为内核开发者正在用一种更通用的机制来代替内核命令行处理机制，以实现引导时间和可加载模块参数的注册。在当前情况下，__setup宏声明了几百个参数，然而在新的开发中希望使用内核头文件.../include/linux/moduleparam.h中定义的一系列函数来实现，更值得注意的是使用module_param*宏系列。这些内容将在第8章中介绍设备驱动程序的时候详细介绍。

上面所说的这种新机制通过在解析程序中包含一个未知的函数指针参数进而保持了向后兼容性，因此，对于module_param*结构来说，是未知的参数就会被视为未知参数，并且对命令行的处理过程就在开发者的控制下重新回到了原有的机制。在仔细研究../kernel/params.c中的代码和../init/main.c中的parse_args()调用后就可以对这一过程有很好的理解。

对于由__setup宏所创建的结构obs_kernel_param，其中标志(flag)成员的用途是最后要注意的内容。仔细研究代码清单5-6就会明白。该结构中称为early的标志用来指示这个特定的内核命令行参数是否会在引导过程中预先使用，一些命令行参数就是特意要在引导过程中提前用到，那么在这种情况下的标志就会为提前解析命令行参数提供一种实现机制。你会在main.c代码中看到一个名为do_early_param()的函数，该函数会遍历数组，该数组是__setup宏结构由目标链接器产生的，同时该函数会处理每一个被标记为预先使用的内核命令行参数，在引导过程执行这一处理操作时给开发者一些控制权。

5.4 子系统初始化

许多Linux子系统的初始化代码都可在main.c中找到。一些子系统的初始化代码在main.c中显而易见，如对init_timers()和console_init()的调用，它们在初始化过程之初就要调用。另外一些子系统所采用的初始化机制与前面所提到的__setup宏非常类似，简单地讲，目标代码链接器会为不同的初始化程序创建函数指针列表，同时采用简单的循环机制依次执行。代码清单5-7显示了这一过程。

代码清单5-7 初始化程序示例

```
static int __init customize_machine(void)
{
    /* customizes platform devices, or adds new ones */
    if (init_machine)
        init_machine();
    return 0;
}
arch_initcall(customize_machine);
```

这部分代码来源于.../arch/arm/kernel/setup.c，它是为一个特殊开发板提供用户定制

的简单程序。

*_initcall 宏

对于代码清单5-7中的初始化程序，有两个要点需要注意。首先，程序中的函数是由__init宏定义的，就像在前面看到的。__init宏将该函数放到了vmlinux ELF文件中一个称为.init.text的段中，我们可以想到将一个函数放到目标文件中一个特殊段中的目的，这是为了当函数不再使用后将函数所占用的内存空间释放。

第二个需要注意的事情是在函数定义之后的宏，即arch_initcall(customize_machine)，该宏是在.../include/linux/init.h中所定义的一系列宏中的一个。这些宏如代码清单5-8所示。

代码清单5-8 initcall宏系列

```
#define __define_initcall(level,fn) \
    static initcall_t __initcall_##fn __attribute_used__ \
    __attribute__((__section__(".initcall" level ".init"))) = fn

#define core_initcall(fn)          __define_initcall("1",fn)
#define postcore_initcall(fn)      __define_initcall("2",fn)
#define arch_initcall(fn)          __define_initcall("3",fn)
#define subsys_initcall(fn)        __define_initcall("4",fn)
#define fs_initcall(fn)            __define_initcall("5",fn)
#define device_initcall(fn)        __define_initcall("6",fn)
#define late_initcall(fn)          __define_initcall("7",fn)
```

__initcall宏与前面介绍的__setup宏在形式上非常相似，这些宏基于函数名声明了一个数据列表，并且使用段属性将这些数据内容放到vmlinux ELF文件中被唯一命名的段中。这样做的好处是，main.c可以任意调用其并不知道的子系统初始化程序，如果不这样做，那么唯一的方法就是采用前面描述的方法，即只能改写main.c中的相关内容，让内核了解每一个子系统。

如代码清单5-8所示，这些段的名称为.initcallN.init，这里的N表示的是数量1~7，数据被分配到由宏命名的函数地址处。在代码清单5-7和代码清单5-8所示的例子中，数据的分配形式如下（为了简化起见，省去了段属性）：

```
static initcall_t __initcall_customize_machine = customize_machine;
```

该数据被放到内核目标文件中的一个名为.initcall11.init的段中。

这里的N用来提供初始化调用的顺序关系，比如使用core_initcall()宏声明的函数在其他所有函数之前被调用，使用postcore_initcall()宏声明的函数在其后被调用，依次类推，使用late_initcall()宏声明的初始化函数在最后被调用。

和__setup宏系列非常类似，*_initcall宏系列可以看作是内核子系统初始化程序的注册函数，而且这些初始化程序也是在内核启动后就要执行，且执行后不再使用。这些宏提供了一种机制，以实现在系统启动过程中可以执行初始化程序，并且在程序执行之后将程序丢弃同时回收内存。在执行初始化程序的时候也为开发者提供了7种不同的级别，因此，如果一个子系统依赖

于另一个子系统可用,那么就可以使用这些级别来提高它的执行顺序。如果使用grep命令查找内核中的[a-z]*_initcall字符串信息,就会发现这些系列的宏在内核中使用非常广泛。

对于*_initcall系列的宏,最后要注意的是:多级别的用法在Linux 2.6内核的开发过程中引入,早期版本的内核是用__initcall()宏来实现的,目前__initcall()宏仍然在广泛使用中,尤其是在设备驱动程序中。为了保持向后兼容性,已经将__initcall()宏定义为device_initcall(),这是一个级别为6的initcall。

5.5 init 线程

../init/main.c中的内容主要用来实现内核的运转。在start_kernel()函数通过调用一些初始化函数执行一些基本的内核初始化任务之后,就产生了第一个内核线程。该线程最终成为内核的init()线程,其线程ID号(PID)为1。可以知道,init()就成为用户空间中所有Linux进程的父进程。在引导过程中运行着两个截然不同的线程:一个是前面提到的start_kernel();另一个就是现在的init()。前者在完成自身的任务之后最终成为idle进程,而后者称为init进程,如代码清单5-9所示。

代码清单5-9 内核init线程的创建

```
static void noinline rest_init(void)
{
    __releases(kernel_lock)

    kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND);
    numa_default_policy();
    unlock_kernel();
    preempt_enable_no_resched();

    /*
     * The boot idle thread must execute schedule()
     * at least one to get things moving:
     */
    schedule();

    cpu_idle();
}
```

从代码清单5-9可以看出, start_kernel()函数调用了rest_init(),通过调用kernel_thread().init来产生内核的init进程,以继续完成内核其余的初始化任务,而由start_kernel()开始的线程在调用cpu_idle()的过程中不停地重复执行。

这样的结果非常有趣。你或许也注意到这个相当庞大的start_kernel()函数被__init宏所标记,这意味着它所占用的内存空间将在内核初始化的最后阶段被释放。在释放内存之前需要退出该函数和它所占用的地址空间,这是通过start_kernel()调用rest_init()来实现的,如代码清单5-9所示,一段非常小的内存空间处在了空闲状态。

5.5.1 通过 `initcall` 初始化

当创建`init()`之后，它会调用`do_initcalls()`函数，而`do_initcalls()`函数是用来调用所有被`*_initcall`宏系列所注册的初始化函数的，其实现代码如代码清单5-10所示。

代码清单5-10 使用`initcalls`初始化

```
static void __init do_initcalls(void)
{
    initcall_t *call;

    for( call = &__initcall_start; call < &__initcall_end; call++) {

        if (initcall_debug) {
            printk(KERN_DEBUG "Calling initcall 0x%p", *call);
            print_symbol(":%s()", (unsigned long) *call);
            printk("\n");
        }

        (*call)();
    }
}
```

除了两个用于指示循环范围的标签`__initcall_start`和`__initcall_end`之外，该段代码很好理解。在C源代码和头文件中不会看到这样的标签，它们是在`vmlinux`链接阶段所用的链接脚本文件中定义的，用来表示使用`*_initcall`宏系列所生成的初始化函数列表的起始和结束位置。你可以在Linux内核顶层目录下的`System.map`文件中看到每一个这样的标签，这些标签以字符串`__initcall`开始，就像代码清单5-8中所表示的那样。

你如果对`do_initcalls()`函数中的调试打印信息感到疑惑的话，可以看一下由在引导过程中设置的内核命令行参数`initcall_debug`所执行的系统调用，该命令行参数允许打印如代码清单5-10所示的调试信息。内核只需简单地以内核命令行参数`initcall_debug`开始就可以实现这些调试信息的输出^①。

下面是一个启用了这些调试语句时的输出的例子：

```
...
Calling initcall 0xc00168f4: tty_class_init+0x0/0x3c()
Calling initcall 0xc000c32c: customize_machine+0x0/0x2c()
Calling initcall 0xc000c4f0: topology_init+0x0/0x24()
Calling initcall 0xc000e8f4: coyote_pci_init+0x0/0x20()
PCI: IXP4xx is host
PCI: IXP4xx Using direct access for memory space
...
```

注意在代码清单5-7中对`customize_machine()`的调用，调试信息的输出包括了函数的虚拟

① 你或许要在降低默认日志级别以后才可以看到这些调试信息，这在许多有关Linux管理的参考手册中都有描述。无论如何，你都可以在内核的日志文件中看到这些信息。

内核地址（在该例中是0xc000c32c）和函数大小（在这里是0x2c）。这是了解内核初始化的一个有效方法，特别是对不同子系统和模块的调用次序的理解。即使是在一个具有相当配置的嵌入式系统之上，也有几十个这样的初始化函数通过这种方式调用。在这个以嵌入式ARM XScale为平台的例子中，共有92个这样不同的内核初始化程序。

5.5.2 引导的最后步骤

在已经创建了init()线程并且对所有不同的初始化调用完成之后，内核开始执行引导过程最后阶段的内容，这些内容包括释放由初始化函数和数据所用的内存资源，打开系统的控制台设备，启动用户空间下的第一个进程。代码清单5-11再现了内核main.c中init()函数最后所执行的过程。

代码清单5-11 main.c中内核引导的最后过程

```
if (execute_command) {
    run_init_process(execute_command);
    printk(KERN_WARNING "Failed to execute %s. Attempting "
                    "defaults...\n", execute_command);
}

run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");

panic ("No init found. Try passing init= option to kernel.");
```

注意，init()函数末尾处的代码：内核的一个panic结果。如果你做过嵌入式系统相关试验或定制过根文件系统，会毫无例外地在用户的控制台输出见到这样非常常见的错误信息，这也是在多个与Linux和嵌入式系统相关的常见问题解答（FAQ）中最容易被问到的问题。

不论怎么讲，这些run_init_process()命令必须要能够正确地执行。函数run_init_process()不会返回调用成功的信息。对于使用新进程代替当前进程的相关内容已经讲了很多，实际中是使用熟悉的execve()系统调用来实现这一功能的。作为userland^①初始化程序的/sbin/init，我们将在下一章中深入研究。

对于嵌入式系统开发者来说，一种选择是使用定制的用户land初始化程序，这是前面代码片段中条件语句的目的。如果执行命令是有效的，它会指向包含用户所提供的命令的字符串，该命令将会在用户空间下执行。开发者在内核命令行中指定该命令，并且通过在本章前面描述的__setup宏来设置。下面是一个内核命令行使用的例子，该内核命令行中包含了在本章中讨论的几个要点；该命令行语句如下：

```
initcall_debug init=/sbin/myinit console=ttyS1,115200 root=/dev/hda1
```

上面这个内核命令行指示内核显示所有在内核中遇到的初始化程序，配置默认控制台设备

^① userland是用户空间下针对程序、库、脚本和其他内容的一个常用术语。

/dev/ttyS1/的波特率为115kbit/s，同时执行用户空间下定制的初始化程序myinit，该程序位于根文件系统下的/sbin目录中，它会指导内核从设备/dev/hda1（硬盘设备的第一个分区）上挂载它的根文件系统。需要注意的是，在通常情况下，给到内核命令行参数的顺序并不是固定的。下一章将会详细介绍用户空间下的系统初始化。

5.6 小结

- Linux内核项目庞大而复杂。理解最后生成映像文件的结构和组成是掌握如何定制嵌入式项目的关键所在。
- 许多体系结构的处理器通过在内核二进制映像文件之上链接一个与体系结构相关的第二阶段引导装入程序来为Linux内核设置适当的运行环境，对于这个与内核功能不同的第二阶段引导装入程序，我们给出了创建步骤。
- 理解内核的初始化流程有助于加深你对Linux内核的理解，并且为你如何实现开发中的特定需求提供了一套参考方法。
- 我们在head.o中找到了内核的入口点，同时也在第一个内核C文件main.c中了解了内核的初始化流程。我们还看到了一个引导中的系统所产生的引导信息，进而对这些重要的初始化内容有了总体的认识。
- 介绍了内核命令行处理过程以及用来声明和处理内核命令行参数的机制，为了实现使用目标链接器生成列表来任意调用设置程序的目的，采用了先进的编码技巧，在本章也做了详细介绍。
- 在内核引导的最后阶段就会产生用户空间的第一个进程。理解这一机制和实现过程有助于你定制和排查在嵌入式Linux系统的启动流程中所遇到的问题。

参考资源

GNU 编译器文档：

<http://gcc.gnu.org/onlinedocs/gcc>^①

使用GNU链接工具LD：

<http://www.gnu.org/software/binutils/manual/ld-2.9.1/ld.html>

内核文档：

.../Documentation/kernel-parameters.txt

① 特别是关于函数属性、类型属性和变量属性的内容。



本章内容

- 根文件系统
- 内核的最后引导过程
- init进程
- 初始RAM磁盘
- 使用initramfs
- 关机
- 小结

第2章曾经指出，Linux内核本身只是嵌入式Linux系统的一小部分。Linux内核完成初始化之后，必须挂载一个根文件系统，并执行由开发人员定义的一系列初始化例程。在本章中，我们将仔细研究内核初始化之后的系统初始化内容。

我们先来看看根文件系统的内容及其布局，接下来开发和研究一个最小的系统配置。本章后面会在这个最小的系统配置中增加一些功能以实现一个嵌入式系统配置的示例。介绍初始RAM磁盘（或initrd）及其操作和用法之后，就完成了对系统初始化的介绍。最后简要介绍了Linux系统关机（shutdown）操作。

6.1 根文件系统

在第5章中，我们研究了初始化过程中Linux内核的行为，还提供了几种挂载根文件系统的方法。像其他高级操作系统一样，Linux也需要一个根文件系统才能体现其卓越的性能。尽管在没有文件系统的环境下使用Linux不存在问题，不过这么做不切实际，因为这样一来就无法充分利用Linux最重要的特性和价值，这就如同要将用户的整个系统应用塞进一个巨大的设备驱动程序或内核线程里。

根文件系统是指作为文件系统层次结构之基部而挂载的文件系统，简言之即“/”。正如第9章将介绍的，即使是一个很小的嵌入式系统也会在文件系统层次结构的不同位置挂载多个文件系统，例如第9章介绍的proc文件系统。proc文件系统是挂载到根文件系统下/proc位置的一个专用文件系统。实际上，根文件系统不过是挂载到文件系统层次结构基部的第一个文件系统。

稍候你会看到，Linux系统对根文件系统有一些特殊的要求，期望根文件系统包含若干程序和实用工具，用于引导系统、初始化网络和系统控制台等服务、加载设备驱动和挂载另外的文件系统。

6.1.1 FHS

一些内核开发人员制定了一个标准，用来规定UNIX文件系统的组织和布局。FHS（File System Hierarchy Standard，文件系统层次结构标准）为实现Linux各个发行版和应用程序之间的兼容性建立了最低基线，本章结尾的“参考资源”中给出了关于FHS的参考资料。如果你想深入了解UNIX文件系统组织的布局和基本原理，建议阅读FHS标准。

许多Linux发行版的文件系统布局方式与FHS标准严格匹配。FHS标准在不同的UNIX或Linux发行版之间提供了一个通用的基本要素，FHS标准允许用户的应用软件（和开发人员）预先获知某些特定系统元素（包括文件和目录）在文件系统上的具体位置。

6.1.2 文件系统布局

出于空间的考虑，许多嵌入式系统开发人员会在一个可引导设备（例如Flash）上创建一个非常小的根文件系统，然后从另一个设备上挂载一个更大的文件系统，这里的设备可能是硬盘也可能是网络中的一个NFS服务器。事实上，在一个最初较小的根文件系统上挂载一个较大的根文件系统并不少见，在本章后面研究初始RAM磁盘的相关内容时，你会看到这样一个例子。

一个简单的Linux根文件系统可能包含如下顶层目录项：

```
.
|--bin
|--dev
|--etc
|--lib
|--sbin
|--usr
|--var
|--tmp
```

表6-1描述了上述各根目录项里最常见的内容。

表6-1 根文件系统顶层目录

目 录	内 容
bin	可执行二进制文件，系统下的所有用户 ^① 都可用
dev	设备节点（参见第8章）
etc	本地系统配置文件
lib	系统库所在目录，如C标准库及其他大量库

^① 嵌入式系统往往除一个根用户外再没有其他用户。

(续)

目 录	内 容
sbin	二进制可执行文件，通常仅限系统超级用户使用
usr	应用程序的辅助文件系统层次结构，通常为只读
var	包含多变的文件，如系统日志和临时配置文件
tmp	临时文件

通过斜杠 (/) 字符可以引用Linux文件系统层次结构的最顶层。例如，要列出根目录的内容，可以键入如下命令：

```
$ ls /
```

输出结果类似如下：

```
root@coyote:/# ls /
bin dev etc home lib mnt opt proc root sbin tmp usr var
root@coyote:/#
```

这个目录列表包含了用于其他用途的目录项，包括/mnt和/proc。注意，我们通过在前面加“/”引用这些目录项，表示这些顶层目录的路径是从根目录开始的。

6.1.3 最小文件系统

为了具体地说明一个根文件系统所必需的内容，我们创建了一个最小根文件系统。这个例子是在一块使用XScale处理器的ADI Coyote评估板上实现的，代码清单6-1为使用tree命令显示这个最小根文件系统的内容。

代码清单6-1 最小根文件系统的内容

```
.
|-- bin
|   |-- busybox
|   '-- sh -> busybox
|-- dev
|   '-- console
|-- etc
|   '-- init.d
|   '-- rcS
'-- lib
    |-- ld-2.3.2.so
    |-- ld-linux.so.2 -> ld-2.3.2.so
    |-- libc-2.3.2.so
    '-- libc.so.6 -> libc-2.3.2.so

5 directories, 8 files
```

该根文件系统配置使用了busybox，busybox是一个非常流行而且名符其实的嵌入式系统工

具包。简言之, busybox是一个独立的二进制可执行文件, 提供对大量常用Linux命令行实用工具的支持。busybox和嵌入式系统的关系非常紧密, 本书专门抽出一章(第11章)详细介绍这个灵活便捷的实用工具。

注意, 代码清单6-1中的这个最小根文件系统例子只有5个目录, 共8个文件。这个极小的根文件系统可以实现系统引导, 并通过串行控制台为用户提供一个功能完整的命令行提示, 用户可以使用busybox^①中启用的任意命令。

从/bin目录开始看, 在其下面已经有了可执行的busybox文件和指向busybox的软链接(soft link) sh, 你一会儿就会明白这样做的必要性。/dev目录下的文件是打开一个控制台进行输入/输出所需要的设备节点(device node)^②。/etc/init.d目录下的rcS文件是由busybox启动时处理的默认的初始化脚本文件, 尽管该文件并不是必需的。使用rcS文件之后就不会出现busybox发出的警告信息, 该警告信息只会在rcS文件缺失的时候才出现。

上述根文件系统必需的最后两个目录项及两个文件是两个库: GLIBC(libc-2.3.2.so)库和Linux动态加载器(ld-2.3.2.so)。GLIBC包括C标准库函数(如printf())以及绝大多数应用程序所依赖的其他大量库函数; Linux动态加载器用于将二进制可执行文件加载到内存中, 并且执行应用程序引用所需共享库函数的链接工作。这里包括的两个软链接是指向ld-2.3.2.so的ld-linux.so.2和指向libc-2.3.2.so的libc.so.6, 这些链接使这些共享库不受版本影响并且具有向后兼容的特性, 在所有Linux系统下都能看到这类链接。

这个简单的根文件系统实现了一个功能完整的系统。以本书所用ARM/XScale为例的开发板进行试验后发现, 这个小型根文件系统的大小为1.7 MB, 而且有趣的是, 该根文件系统超过80%的大小为C库所占用。如果你需要为嵌入式系统进一步缩小C库的大小, 可以参考<http://libraryopt.sourceforge.net/>上的库优化工具(Library Optimizer Tool)。

6.1.4 根文件系统带来的挑战

根文件系统带给嵌入式系统的挑战很容易解释, 但它所带来的问题却并不容易解决。除非用户足够幸运, 拥有相当大的硬盘或闪存设备来开发嵌入式系统, 否则就会发现, 很难将所有用户应用程序和工具放进一个单独的闪存介质。尽管闪存介质的成本在不断下降, 但是需要缩减产品成本并加速产品上市时间的竞争压力始终存在。作为嵌入式操作系统, Linux日益流行的一个最大原因, 在于Linux具有数量巨大且仍在不断增加的应用软件。

对根文件系统进行裁减以便塞进一个给定大小的存储空间可能非常棘手。许多软件包和子系统包含了几十甚至几百个文件。除了应用程序之外, 许多软件包还包括配置文件、库、配置工具、图标文件、文档文件、国际化相关的区域文件、数据库文件, 等等, 不一而足。由Apache软件基金会主导开发的Apache网络服务器是嵌入式系统下应用广泛的程序之一。某个非常流行的嵌入式系统下的Apache基本软件包包含了254个不同的文件, 此外, 这些文件并不是简单地复制到用户

^① busybox命令将在第11章中介绍。

^② 设备节点的具体内容将在第8章详细介绍。

文件系统下某个目录中，要想不经修改就能运行Apache应用程序，就需要将它们安置到文件系统下几个不同的地方。

这些是发行工程（distribution engineering）的一些基础概念，它们可能非常单调乏味。Linux发行商，如Red Hat（在桌面版和企业版市场）和Monta Vista Software（在嵌入式市场），在以下方面投入了大量工程资源：将大量的程序、库、工具和应用程序打包在一起，组合成一个Linux发行版。构建一个根文件系统必然会使用较小规模的版本发行工程要素。

6.1.5 试错法

直到现在，移植用户根文件的唯一方法仍然是采用试错法（Trial-and-Error Method）。或许这一过程可以通过创建一系列与此相关的脚本来自动实现，但是某个给定功能需要哪些文件，仍要取决于开发人员所掌握的知识。诸如Red Hat软件包管理器（Red Hat Package Manager, rpm）之类的工具可以用来在用户的根文件系统中安装软件包。rpm在给定的软件包内具有合理的依赖性，但是它比较复杂而且难于学习。此外，采用rpm无法轻易创建小型根文件系统，因为它本身并不能够在安装过程中剔除给定软件包里诸如描述文档和用不到的程序等不必要的文件。

6.1.6 自动化文件系统构建工具

居于领先地位的嵌入式Linux发行版，为实现在闪存或其他存储介质中自动构建根文件系统，推出了功能强大的工具，这些工具通常都带有良好的操作界面，使得开发人员可以轻松选择应用程序或功能所需的文件。它们具有从一个软件包剔除无用文件（如文档或其他不需要文件）的功能，而且很多这样的工具具有选择单个文件的功能。这些工具也可以为用户在所选择设备上进行的后续安装提供不同形式的文件系统。想要了解这些功能强大的工具，用户可以与所熟悉的嵌入式Linux发行版开发商联系。

6.2 内核的最后引导过程

前一章已经介绍了在系统引导最后阶段的内核动作，为方便起见，代码清单6-2列出了位于.../init/main.c的这最后一部分代码。

代码清单6-2 main.c中的最后引导过程

```
...
    if (execute_command) {
        run_init_process(execute_command);
        printk(KERN_WARNING "Failed to execute %s. Attempting "
                        "defaults...\n", execute_command);
    }

    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");

    panic("No init found. Try passing init= option to kernel.");
```

在引导的最后阶段，内核会产生一个名为init的内核线程，代码清单6-2正是该内核线程执行的最后一系列事件。其中，`run_init_process()`是一个小巧的`execve()`函数的包装器^①，后者是一个系统调用，其行为相当有趣。如果在该函数调用过程没有发生错误，`execve()`就不会返回。调用（该函数的）线程执行所在的内存空间由被调用程序的内存映像覆写（`overwrite`）。实际上，被调用程序会直接取代调用线程，包括继承其进程ID（PID）。

在Linux内核开发过程中，这样的初始化流程模式保持了很长时间。其实，1.0版本的Linux包含一个类似的模式，本质上，这是用户空间^{*}处理过程的开始。正如代码清单6-2所示，除非Linux内核成功执行了四个程序中的一个，否则内核即告停止（`halt`），同时显示传入`panic()`系统调用的信息。如果你从事嵌入式开发已经有了一段时间，特别是有过根文件系统上的开发经验，就会非常熟悉上面的内核`panic()`及其产生的信息！在Google上搜索`panic()`产生的错误信息，你会看到关于该问题一页又一页的FAQ。当你掌握了本章的内容之后，就会成为擅长处理这个常见问题的专家。

注意这些程序的一个关键点：它们都必须位于根文件系统中的特定位置，其结构与代码清单6-1相仿。因此，我们至少要满足Linux内核对于init程序的要求，也即init程序在其环境中是可执行的。

看一下代码清单6-2，这意味着至少有一个`run_init_process()`函数调用必须成功执行，此外内核会试图按一定次序执行这四个程序。正如上面的代码清单所示，如果这四个程序没有一个执行成功，启动中的内核就会调用可怕的`panic()`函数并立即终止。需要记住的是，`.../init/main.c`中的这部分代码在引导阶段只执行一次，如果执行没有成功，内核除了通过调用`panic()`函数进行报错并终止引导过程之外，几乎不再有任何动作。

6.2.1 用户空间下第一个程序

在绝大部分Linux系统中，内核在启动过程中会执行`/sbin/init`，这也正是代码清单6-2中首先尝试执行`/sbin/init`的原因。实际上，这也成了用户空间下第一个要运行的程序，可以回顾下面的引导次序：

- (1) 挂载根文件系统；
- (2) 创建第一个用户空间下的程序，这里即是init。

在代码清单6-1所示的最小根文件系统中，前三次创建用户空间程序的尝试都没有成功，因为在这个文件的任何地方都找不到一个名为init的可执行文件。回顾代码清单6-1，我们有一个名为sh的软链接，指向可执行文件busybox。现在你应该意识到这个软链接的作用了：它会使Linux将busybox作为初始化进程加以执行，同时也满足了在用户空间中的一个可执行shell的一

^① 实际上，因为一些特殊的活动，现代Linux内核在引导过程的早期已经创建了类似于用户空间的环境，此内容已经超出了本书的范围。

^② 现今的Linux内核在引导过程的早期阶段由于特定需要而创建了一个类用户空间的环境，这不在本书讨论范围。

般要求^①。

6.2.2 解决依赖

简单地将一个init这样的可执行程序添加到用户文件系统中并期望系统引导正常,还远远不够。对于添加到根文件系统的每一个程序,你还必须满足该程序的依赖。绝大部分程序有两种依赖:解析动态链接可执行程序的未定参考(unresolved reference)所需的文件(库);以及应用程序可能需要的外部配置文件或数据文件。对于前者,我们可以用一个工具找出相关文件;而对于后者,只能通过大致了解出现问题的应用程序进行确认。

举个例子有助于理解上述内容。init进程是一个动态链接的可执行程序。要运行init进程,我们需要提供它依赖的相关库,为此专门开发了一个工具:ldd。要明确一个给定程序依赖哪些库,只需对这个二进制文件直接执行交叉版本的ldd命令:

```
$ ppc_4xxFP-ldd init
      libc.so.6 => /opt/eldk/ppc_4xxFP/lib/libc.so.6
      ld.so.1 => /opt/eldk/ppc_4xxFP/lib/ld.so.1
$
```

从这个ldd命令输出可以看到,例子中的PowerPC init可执行文件依赖于两个库,即C标准库(libc.so.6)和Linux动态加载器(ld.so.1)。

要满足可执行文件的第二种依赖,即它可能需要的配置文件和数据文件,需要知道这个子系统是如何工作的。举个例子,init进程要从/etc目录下一个称为inittab的数据文件中读取运行配置,除非你使用的是内置了这些信息的工具,例如6.1.6节描述的那些工具,否则就必须自行提供这些信息。

6.2.3 定制初始化程序

值得注意的是,对于启动过程中所执行的初始化程序,开发人员是可以控制的,其实现方法就是通过使用内核命令行参数。由代码清单6-2中对panic()函数调用所用到的字符串信息可以得到一些提示,下面的命令可能就是开发人员指定init进程时会用到的:

```
console=ttyS0,115200 ip=bootp root=/dev/nfs init=/sbin/myinit
```

要用这种方式在内核命令行中指定init=,必须在/sbin目录下提供一个名为myinit的二进制可执行文件,这将是内核引导过程结束之际取得系统控制权的第一个进程。

6.3 init 进程

除非你要做一些非常特殊的配置,否则永远都不需要一个由用户定制的初始化程序,因为标准init进程的功能和使用非常灵活。init进程以及即将探讨的一系列启动脚本一起实现了通常所谓的System V Init,这一名称源自最初使用这种模式的UNIX System V。现在我们就来研究这个功能强大的系统配置和控制工具。

^① 当busybox被符号链接sh所调用的时候,它会创建一个shell,我们将在第11章中详细介绍。

在前几节提到，init进程是在内核引导过程结束后由内核产生的第一个用户空间程序。就像你所了解的，在运行中的Linux系统里，每个进程与其他进程存在“子父”关系，而init进程是Linux系统里所有用户空间进程最终的父进程。此外，init进程会提供一套环境变量（包括PATH和CONSOLE等）的默认配置，供其他所有进程继承。

init进程的主要作用是通过一个特殊的配置文件来产生其他程序，这个特殊的配置文件通常就是/etc/inittab。Init具有运行级（runlevel）的概念，运行级可以视作系统的一种运行状态。每个运行级都由进入该运行级时启用的服务和产生的程序所确定。

init进程在任何给定时刻都只能处于一种运行级。init进程使用的运行级包含从0到6的运行级和一个称为s的特殊运行级。运行级0指示init进程终止系统，而运行级6则会使系统重新启动。对于每一种运行级，通常都会提供一套启动和关闭脚本，以便指定每种运行级的系统应当执行的动作。每个指定运行级所执行的动作都由稍候将介绍的配置文件/etc/inittab决定。

许多Linux版本都保留了一些运行级用于特定的目的，表6-2详细列出了在许多Linux版本中的init运行级及其使用目的。

表6-2 运行级

运 行 级	使用目的
0	关闭系统（停机）
1	维护用的单用户系统配置
2	用户自定义
3	通用多用户配置
4	用户自定义
5	多用户模式，启动进入图形用户界面
6	重新启动系统（reboot）

运行级脚本通常位于/etc/rc.d/init.d目录下，从中可以找到绝大部分的脚本，分别用于启用或禁用对应的服务。调用某个脚本并传入一个合适的参数，如start（启动）、stop（停止）或restart（重新启动），就能对相应的服务进行手动配置。代码清单6-3例举了如何重新启动NFS服务。

代码清单6-3 重新启动NFS服务

```
$ /etc/rc.d/init.d/nfs restart
Shutting down NFS mountd:      [ OK ]
Shutting down NFS daemon:      [ OK ]
Shutting down NFS quotas:      [ OK ]
Shutting down NFS services:    [ OK ]
Starting NFS services:         [ OK ]
Starting NFS quotas:           [ OK ]
Starting NFS daemon:           [ OK ]
Starting NFS mountd:           [ OK ]
```

如果你用过桌面Linux发行版（如Red Hat或Fedora），一定会在系统启动过程中看到过类似代

码清单6-3所示的信息。

运行级是由该运行级下启用的服务所定义的。大多数Linux发行版都会在/etc下提供一个目录结构，其中包含指向目录/etc/rc.d/init.d中服务脚本的链接。这些运行级相关的目录一般位于/etc/rc.d中。在这个目录里，你会看到一系列运行级的目录，分别包含各个运行级里启动和停止服务的脚本，init只是在进入和退出运行级时执行这些脚本。这些脚本定义了系统的状态，同时，inittab文件会告知init关于脚本和指定运行级的对应关系。代码清单6-4显示了一个/etc/rc.d下的目录结构，这些目录控制着运行级所分别对应服务的启动和停止。

代码清单6-4 包含运行级的目录结构

```
$ ls -l /etc/rc.d
total 96
drwxr-xr-x 2 root root 4096 Oct 20 10:19 init.d
-rwxr-xr-x 1 root root 2352 Mar 16 2004 rc
drwxr-xr-x 2 root root 4096 Mar 22 2005 rc0.d
drwxr-xr-x 2 root root 4096 Mar 22 2005 rc1.d
drwxr-xr-x 2 root root 4096 Mar 22 2005 rc2.d
drwxr-xr-x 2 root root 4096 Mar 22 2005 rc3.d
drwxr-xr-x 2 root root 4096 Mar 22 2005 rc4.d
drwxr-xr-x 2 root root 4096 Mar 22 2005 rc5.d
drwxr-xr-x 2 root root 4096 Mar 22 2005 rc6.d
-rwxr-xr-x 1 root root 943 Dec 31 16:36 rc.local
-rwxr-xr-x 1 root root 25509 Jan 11 2005 rc.sysinit
```

每一个运行级都由rcN.d里包含的脚本进行定义，其中N即为运行级。在每个rcN.d目录下，你可以看到大量以特定顺序排列的符号链接，这些符号链接的名称都以K或S开头。以S开头的服务脚本在执行启动操作时调用，而以K开头的服务脚本则在执行停止操作时调用。代码清单6-5例举了一个只包含几个服务脚本的目录。

代码清单6-5 运行级目录示例

```
lrwxrwxrwx 1 root root 17 Nov 25 2004 S10network -> ../init.d/network
lrwxrwxrwx 1 root root 16 Nov 25 2004 S12syslog -> ../init.d/syslog
lrwxrwxrwx 1 root root 16 Nov 25 2004 S56xinetd -> ../init.d/xinetd
lrwxrwxrwx 1 root root 16 Nov 25 2004 K50xinetd -> ../init.d/xinetd
lrwxrwxrwx 1 root root 16 Nov 25 2004 K88syslog -> ../init.d/syslog
lrwxrwxrwx 1 root root 17 Nov 25 2004 K90network -> ../init.d/network
```

在这个例子中，我们指示启动脚本在进入这个假想的运行级时开启3个服务：network、syslog和xinetd。由于以S*开头的脚本会在S*后面紧跟着数字，所以它们会按照这些数字的顺序来执行。类似地，在退出该运行级的时候，xinetd、syslog和network这3个服务依次被终止，而且也会类似地根据符号链接名中字母K后面的两位数字的大小顺序依次将这些服务停止。在真正的系统中，毫无疑问，目录下的脚本（链接）数量要多得多。你还可以为自己特定的应用程序添加相应的服务脚本。

执行这些服务启动和关闭脚本的顶层脚本则在init配置文件中定义，接下来就将探讨这个配置文件。

6.3.1 inittab

启动init进程时会试着读取系统配置文件/etc/inittab，包含针对每个运行级及适用于所有运行级的指令。这个文件和init的行为在绝大多数Linux工作站的帮助手册里都有详细的记录，在一些关于Linux系统管理的书籍中也有详细描述。我们不打算在这里重复这些工作，而是重点着眼于开发人员如何为嵌入式系统配置inittab。至于inittab和init如何一起工作的详细说明，在大部分Linux工作站下可以通过键入man init和man inittab查看帮助手册得知。

我们来看一个简单嵌入式系统的典型inittab文件。代码清单6-6是针对一个系统的inittab简单示例，该文件只支持一个运行级，实现系统的关机和重启。

代码清单6-6 简单的inittab文件示例

```
# /etc/inittab

# The default runlevel (2 in this example)
id:2:initdefault:

# This is the first process (actually a script) to be run.
si::sysinit:/etc/rc.sysinit

# Execute our shutdown script on entry to runlevel 0
l0:0:wait:/etc/init.d/sys.shutdown

# Execute our normal startup script on entering runlevel 2
l2:2:wait:/etc/init.d/runlvl2.startup

# This line executes a reboot script (runlevel 6)
l6:6:wait:/etc/init.d/sys.reboot

# This entry spawns a login shell on the console
# Respawn means it will be restarted each time it is killed
con:2:respawn:/bin/sh
```

这个非常简单的inittab^①脚本描述了3个不同的运行级，每个运行级都与一个脚本相关联，这些脚本必须是开发人员根据每个运行级所期望的动作而创建的。当init进程读取这个文件时，执行的第一个脚本是/etc/rc.sysinit，由标签sysinit表示。然后init进程进入运行级2，执行为运行级2定义的脚本，这个例子里即为脚本/etc/init.d/runlvl2.startup。正如代码清单6-6中的:wait:标签所示，init进程在该脚本执行完毕之前一直处于等待状态。在运行级2的脚本执行完毕后，init进程会在控制台中生成一个shell（通过符号链接/bin/sh），如代码清单6-6最后一行所示。关键词respawn指示init进程一旦检测到shell退出便重新启动shell。代码清单6-7显示了启动期间的输出信息。

代码清单6-7 启动信息示例

...

① 这个inittab对于小型专用的嵌入式系统而言，是一个很好的例子。

```
VFS: Mounted root (nfs filesystem).  
Freeing init memory: 304K  
INIT: version 2.78 booting  
This is rc.sysinit  
INIT: Entering runlevel: 2  
This is runlvl2.startup  
  
#
```

这个例子中的启动脚本除了为便于说明而打印自身被执行的信息外，不做其他任何事情。当然，在一个实际的系统中，这些脚本会启用若干功能和服务，完成一些有用的任务！就该例的这个简单配置文件而言，你可以在脚本/etc/init.d/runlvl2.startup里为特定的组件启用一些服务和应用程序，同时在关机或者重启脚本里执行逆操作，即终止这些应用程序、服务和设备。我们会在下一节分析一些典型的系统配置，以及在启动脚本里启用这些配置所必需的条目。

6.3.2 Web 服务器启动脚本示例

这个启动脚本示例很简单，毕竟只是为了说明其工作机制，并指导你设计自己的系统启动和关机动作。这个例子基于busybox工具，其初始化行为与前面提到的init有稍许差别，具体的差别将在第11章中详细介绍。

在一个包含Web服务器的典型嵌入式设备中，为便于维护和远程接入，我们可能会需要设备提供多个服务器。在这个例子中，我们通过inetd启用了两个服务器，支持HTTP和Telnet接入。代码清单6-8为我们假想的Web服务器设备提供了一个简单的rc.sysinit脚本。

代码清单6-8 Web服务器的rc.sysinit

```
#!/bin/sh  
  
echo "This is rc.sysinit"  
  
busybox mount -t proc none /proc  
  
# Load the system loggers  
syslogd  
klogd  
  
# Enable legacy PTY support for telnetd  
busybox mkdir /dev/pts  
busybox mknod /dev/ptmx c 5 2  
busybox mount -t devpts devpts /dev/pts
```

在这个简单的初始化脚本中，我们首先启用了proc文件系统，第9章会详细介绍这个非常有用的子系统。接着启动了系统日志记录进程，以便捕获系统运行过程中的信息，当系统运行出现某些错误时，它能派上大用场。最后几行启用了UNIX PTY子系统支持，该子系统是本例实现Telnet服务器时所必需的。

代码清单6-9中显示了运行级2的启动脚本里的命令，这些命令启用了操作该设备所需的服务。

代码清单6-9 运行级2的启动脚本示例

```
#!/bin/sh

echo "This is runlvl2.startup"

echo "Starting Internet Superserver"
inetd

echo "Starting web server"
webs &
```

毋庸置疑，这个运行级2的启动脚本非常简单。首先我们启用了所谓的Internet超级服务器inetd，它会拦截常见的TCP/IP请求并为之创建相应的服务。在这个例子中，我们通过名为/etc/inetd.conf的配置文件启用Telnet服务，然后执行Web服务器，本例为webs。这就是该脚本的全部内容，尽管简单，不过这个配置已能支持Telnet和Web服务了。

要完成上述配置工作，还需要提供一个关闭脚本（参考代码清单6-6）。就本例而言，这个脚本会在系统关机之前终止上面的Web服务器和Internet超级服务器。对这个例子来说，要实现正确的系统关机，这样的操作足够了。

6.4 初始 RAM 磁盘

Linux内核提供了一种机制，能挂载一个早期的根文件系统来执行启动相关的系统初始化和配置任务；该机制即为初始RAM磁盘（简称initrd）。相关支持代码必须直接编译译到内核中，该内核配置选项位于内核配置工具的Block Devices，RAM disk support选项下，图6-1显示了initrd的配置界面。

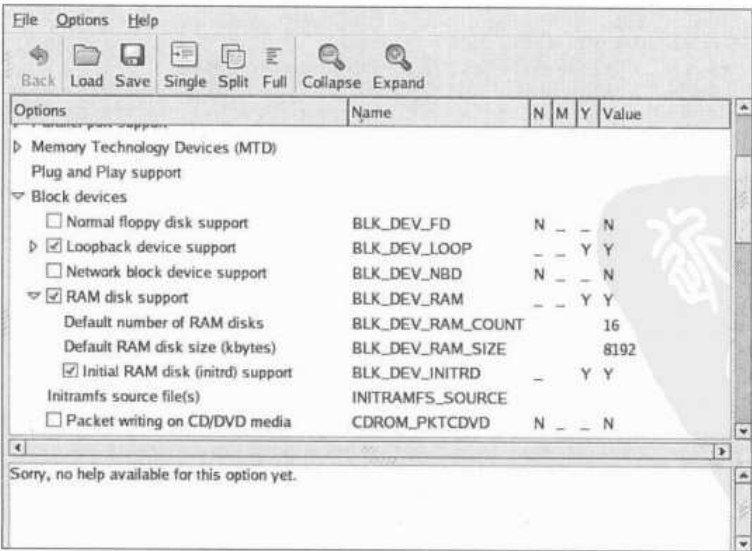


图6-1 Linux内核配置工具

6.4.1 初始 RAM 磁盘的目的

初始RAM磁盘是一个功能完备的小型根文件系统，通常用来在系统引导过程结束之前加载一些特定的设备驱动模块。例如Red Hat和Fedora Core的Linux工作站发行版就使用了初始RAM磁盘，以便在挂载真正的根文件系统之前加载EXT3文件系统相关的设备驱动。Initrd一般用来加载访问真正的根文件系统所必需的设备驱动。

6.4.2 使用 initrd 引导

为了使用initrd的功能，绝大多数体系结构下，引导装入程序会负责将initrd映像传递到内核中。常见的情况是引导装入程序将一个压缩过的内核映像文件加载到内存中，并将initrd映像加载到另一段可用的内存区域中。要实现上述目的，在将控制权移交给内核之前，引导装入程序需要负责将initrd映像文件的加载地址传递给内核。具体的实现机制视体系结构、引导装入程序和平台实现的不同而有所差异；但不管怎样，内核都必须知道initrd映像的具体位置，这样才能进行正确加载。

对于某些体系结构和平台，构建内核时只产生一个复合的二进制映像，这种方案用在引导装入程序并不支持加载initrd映像的情况下。在这种情况下，内核和initrd映像被直接连接在一起。你可以在内核makefile文件中看到对bootpImage这个合成映像的说明，目前，这种方式仅用于arm体系结构。

那么内核又是如何知道哪里可以找到initrd映像呢？除非是通过引导装入程序中某些非常巧妙的地方，它通常是简单地通过内核命令行将initrd映像文件在内存中的起始地址和大小传递给内核来实现的。下面是使用内核命令行传递的一个例子，该例子所基于的平台是一个流行的ARM评估板，它所使用的处理器为TI的OMAP 5912。

```
console=ttyS0,115200 root=/dev/nfs \
nfsroot=192.168.1.9:/home/chris/sandbox/omap-target \
initrd=0x10800000,0x14af47
```

出于排版需要，上面的内核命令行被分成了几行显示；实际使用时，它是单独的一行，各部分之间用空格分隔。该内核命令行定义了如下内核行为：

- 在ttyS0设备上（波特率为115kbit/s）指定一个控制台；
- 通过NFS挂载一个根文件系统；
- 在宿主机192.168.1.9上找到NFS根文件系统，路径为/home/chris/sandbox/omaptarget；
- 从物理内存的0x10800000地址处加载并挂载一个初始RAM磁盘，其大小为0x14AF47（1 355 591字节）。

关于本例需要特别注意一点：几乎所有initrd映像都经过压缩，因此内核命令行里指定的initrd大小是指压缩后的initrd映像文件大小。

6.4.3 引导装入程序对于 initrd 的支持

我们来看一个运行在ARM处理器上的基于U-Boot的简单例子。这个引导装入程序已实现了

对Linux内核的支持。使用U-Boot可以很容易地将initrd映像文件和内核映像文件组合在一起。代码清单6-10显示了一个包含ramdisk映像的典型引导过程。

代码清单6-10 使用ramdisk引导内核

```
# tftpboot 0x10000000 kernel-uImage
...
Load address: 0x10000000
Loading: ##### done
Bytes transferred = 1069092 (105024 hex)

# tftpboot 0x10800000 initrd-uboot
...
Load address: 0x10800000
Loading: ##### done
Bytes transferred = 282575 (44fcf hex)

# bootm 0x10000000 0x10800040
Uncompressing kernel.....done.
...
RAMDISK driver initialized: 16 RAM disks of 16384K size 1024 blocksize
...
RAMDISK: Compressed image found at block 0
VFS: Mounted root (ext2 filesystem).
Greetings: this is linuxrc from Initial RAMDisk
Mounting /proc filesystem

BusyBox v1.00 (2005.03.14-16:37+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

# (<<<< Busybox command prompt)
```

从代码清单6-10中可以简单地了解U-Boot，下一章会更详细地介绍U-Boot。其中，U-Boot使用tftpboot命令从tftp服务器下载内核映像，内核映像下载后被放置到该目标板中内存起始地址256MB（16进制表示为0x10000000）处^①。从tftp服务器下载的第二个映像文件是initrd映像，并放置在内存中较高的地址处（本例为256MB + 8MB地址处）。最后，我们使用了U-Boot的bootm命令（即boot from memory命令），它带有两个参数：前者为Linux内核映像的地址，后者是initrd映像地址。

这里特别关注一下U-Boot的特点。它完全支持通过以太网连接加载内核映像和ramdisk映像，这是一个非常有用的开发配置手段。当然，通过其他方式也能将内核映像和ramdisk映像加载到开发板中，比如可以使用基于硬件手段的闪存编程工具将它们烧写到闪存中，或者通过RS-232下载内核和文件系统的映像文件。但是由于这些映像文件一般都比较大（一个内核映像可能有一兆字节，ramdisk映像有几十兆字节），因此，如果采用基于以太网的tftp下载方式，开发人员就将

^① 我们所用的开发板的物理SDRAM起始地址恰好为256MB。

节省大量的工程时间。不论采用哪种引导装入程序，务必确定它支持通过网络方式下载开发映像文件。

6.4.4 `initrd` 的奥妙所在：`linuxrc` 文件

在内核引导时，它会检测到存在`initrd`映像，将其从RAM中的指定物理地址处将这个二进制压缩格式文件复制到恰当的内核`ramdisk`中，并将其挂载为根文件系统。`initrd`的奥妙之处来自`initrd`映像中某个特殊文件的内容。当内核挂载这个初始`ramdisk`时，它会查找一个名为`linuxrc`的特殊文件；Linux内核会把该文件当作脚本文件，并执行包含在其中的命令。这一机制使得系统设计人员可以定制`initrd`的行为，代码清单6-11列举了一个简单的`linuxrc`文件。

代码清单6-11 `linuxrc`文件示例

```
#!/bin/sh

echo 'Greetings: this is 'linuxrc' from Initial Ramdisk'
echo 'Mounting /proc filesystem'
mount -t proc /proc /proc

busybox sh
```

实际上，该文件会包括在挂载真正的根文件系统之前需要执行的指令。例如，为了从CompactFlash存储设备上获取一个真正的根文件系统，可能需要加载CompactFlash设备的驱动程序。这个例子只是简单地创建一个`busybox sh`，并终止引导过程以进行研究。你可以从代码清单6-10中看到由`busybox sh`给出的提示符`#`。如果在该提示符后面键入`exit`命令，内核将继续其引导过程直到结束。

当内核将`ramdisk`映像从物理内存复制到内核`ramdisk`时，内核会释放原来`ramdisk`映像所占用的内存空间，你可以把它看作是从物理内存的实地址处将`initrd`映像文件复制到内核自身的虚拟内存中（以内核`ramdisk`设备的形式存在）的一个过程。

代码清单6-11中最后需要说明的一点是：挂载`/proc`文件系统，该挂载命令中的`proc`单词看似多余；下面的命令同样有效：

```
mount -t proc none /proc
```

注意，上面`mount`命令将设备名部分改为`none`，该挂载命令忽略了设备名描述，是因为并没有实际的物理设备与`proc`文件系统相对应，该命令采用`-t proc`就足以将`/proc`文件系统挂载到`/proc`挂载点。使用前一个命令形式只是为了说明我们正在将一个内核虚拟设备（`/proc`文件系统）挂载到`/proc`挂载点，实际上挂载命令会忽略这个参数（设备名）。你可以选择自己喜欢的做法。

6.4.5 `initrd` 探究

作为Linux引导过程的一部分，Linux内核必须找出并且挂载一个根文件系统。在引导过程的后期，内核通过一个名为`prepare_namespace()`的函数来决定挂载什么并且在哪儿挂载。如果

当前内核如图6-1所示的配置启用了initrd支持，同时Linux内核命令也按这样进行了配置，那么内核就会对压缩的initrd映像文件进行解压缩，同时最终会从物理内存中把该映像文件内的内容复制到一个ramdisk设备（/dev/ram）里。这个时候，我们就在内核的ramdisk下有了一个恰当的文件系统。在将该文件系统的内容读到ramdisk之后，内核会将该ramdisk设备挂载为它的根文件系统。最后，内核生成一个内核线程来执行initrd映像^①中的linuxrc脚本文件。

当执行linuxrc脚本之后，内核会卸载initrd并且继续执行系统引导的最后阶段。如果当前系统就有一个叫作/initrd的根设备，那么Linux会将initrd文件系统挂载到该路径下（在本书中称为挂载点）；如果该目录（/initrd）不存在，那么initrd映像只是简单地被丢弃。

如果内核命令行中包含一个指定ramdisk的命令行参数root=（例如root=/dev/ram0），在前面内容中描述的initrd处理过程就会有两个重要的改变。首先，对于可执行文件linuxrc的处理会跳过；其次，Linux内核不会再试着挂载其他文件系统作为其根文件系统，也就是说你会有这样的一个Linux系统，即以initrd作为唯一的根文件系统。在对系统进行小型化配置的时候这样做是非常有益的。在这样的系统下，唯一的根文件系统即为ramdisk。将/dev/ram0内容放到内核命令行中将使得完全的系统初始化以initrd作为最终的根文件系统而结束。

6.4.6 构建 initrd 映像文件

构建一个合适的根文件系统映像是嵌入式系统开发所面临的挑战之一，而创建一个恰当的initrd映像则更具有挑战性，因为它要求更加简练、更加专业化。基于以上考虑，我们将在本节探讨initrd的要求以及initrd文件系统的内容。

使用tree命令来显示本章这个initrd映像示例中的内容，其内容如代码清单6-12所示。

代码清单6-12 initrd示例内容

```
-- bin
|  |-- busybox
|  |-- echo -> busybox
|  |-- mount -> busybox
|  '-- sh -> busybox
-- dev
|  |-- console
|  |-- ram0
|  '-- ttyS0
-- etc
-- linuxrc
'-- proc

4 directories, 8 files
```

① 出于必要性（写作空间）的考虑，对该过程的描述非常简单，实际原理和文中描述是类似的，但是在本文中仍然略去了对几个重要细节的介绍，建议你参阅相关的内核源代码做更深入的了解，具体为.../init/main.c和.../init/do_mounts*.c。

可以看到,其内容非常简练,解压缩后的所有文件加起来不到500KB。由于该initrd是基于busybox构建的,因此它具有很好的功能。由于busybox是静态链接的,因此不依赖任何系统库。第11章将详细介绍busybox。

6.5 使用 initramfs

initramfs是用来执行早期用户空间程序所采用的一种新机制(在Linux 2.6的内核中使用),它与前面描述的initrd在概念上较为相似,其作用也很相似,即在挂载真正的根文件系统之前加载必要的设备驱动程序。然而,二者在某些方面还是有显著区别的。

initrd和initramfs的具体执行细节有着显著的区别。例如,initramfs要在调用do_basic_setup()^①之前加载,它会在硬件设备的驱动程序加载之前加载其固件程序。如果要了解更具体的内容,可以参阅最新的相关Linux内核文档,详见.../Documentation/filesystems/ramfs-rootfs-initramfs.txt。

从实用角度看,initramfs更易使用。initramfs是一种cpio格式的档案文件,而initrd是一种gzipped格式的系统映像文件,二者这种简单的区别就使得initramfs更容易使用。当用户编译链接Linux内核映像的时候,initramfs会被集成到Linux内核源代码中并且被自动编译,此外,对initramfs做出改变要比构建和加载一个新的initrd映像更容易一些。代码清单6-13显示了Linux内核下.../usr目录的内容,就是在这里编译链接了initramfs映像,代码清单6-13所示的内容为Linux内核编译链接之后的结果。

代码清单6-13 内核initramfs映像的编译链接目录

```
$ ls -l
total 56
-rw-rw-r-- 1 chris chris 834 Mar 25 11:13 built-in.o
-rwxrwxr-x 1 chris chris 11512 Mar 25 11:13 gen_init_cpio
-rw-rw-r-- 1 chris chris 10587 Oct 27 2005 gen_init_cpio.c
-rw-rw-r-- 1 chris chris 512 Mar 25 11:13 initramfs_data.cpio
-rw-rw-r-- 1 chris chris 133 Mar 25 11:13 initramfs_data.cpio.gz
-rw-rw-r-- 1 chris chris 786 Mar 25 11:13 initramfs_data.o
-rw-rw-r-- 1 chris chris 1024 Oct 27 2005 initramfs_data.S
-rw-rw-r-- 1 chris chris 113 Mar 25 11:13 initramfs_list
-rw-rw-r-- 1 chris chris 1619 Oct 27 2005 Kconfig
-rw-rw-r-- 1 chris chris 2048 Oct 27 2005 Makefile
```

该目录下的initramfs_list文件包含了要在initramfs档案中添加的文件列表,对于最新内核版本所默认添加的内容如下所示:

```
dir /dev 0755 0 0
nod /dev/console 0600 0 0 c 5 1
dir /root 0700 0 0
```

^① do_basic_setup会在.../init/main.c中被调用,而它调用的是do_initcalls()函数,设备驱动程序模块初始化程序将在这个过程中调用到,具体细节已在第5章和代码清单5-10中介绍了。

这是一个小型的目录结构，包含了/root和/dev目录以及表示控制台的一个独立设备节点。添加这些文件可以用来构建用户自己的initramfs，用户也可以通过使用内核配置工具来指定创建initramfs的源文件。在内核配置工具中启用INITRAMFS_SOURCE选项并且使它指向用户的开发平台，内核编译链接系统就将使用这些文件作为用户initramfs映像的源文件。

这个编译链接目录生成的最后一个用来构建initramfs映像的文件是initramfs_data_cpio.gz，它是一个包含了用户所指定文件（不论是通过initramfs_list指定还是通过选择内核编译工具的INITRAMFS_SOURCE选项来指定）的压缩档案文件，该档案文件将被链接到最后的内核映像之中。这也是使用initramfs的另一个优势所在，即不需要在引导时加载一个单独的initrd映像。

6.6 关机

有序关机操作在嵌入式系统的设计中一度曾被忽略，不正确的关机操作会影响到系统的启动时间，甚至会导致某些特定类型的文件系统崩溃。由于采用EXT2文件系统类型的系统在意外掉电后的重新启动过程中需要执行fsck（文件系统检查）命令，而执行该命令花费了太多的时间，这也成为了使用EXT2文件系统（桌面Linux系统多年来默认使用的文件系统）最多的抱怨。对于具有大容量磁盘的服务器来说，对几个大的EXT2分区进行正确的fsck操作可能要花费几个小时。

每个嵌入式系统都可能拥有它自己的关机操作策略，不同的策略可能彼此适用也可能不适用。这里的关机操作所指的范围可能从一个完全的System V关机方案到一个简单脚本的挂起或重新启动。Linux下有一些工具可以用来实现关机操作，包括shutdown、halt和reboot命令，当然，所选的体系结构必须支持这些命令才可以用以实现关机操作。

一个用于关机操作的脚本应该可以终止所有用户空间下的程序，最终关闭那些被进程打开的文件。如果init正在使用中，那么执行init 0命令会将系统挂起。通常来说，首先关机进程会向所有进程发送SIGTERM信号，通知它们系统正在执行关机操作。一段短暂的延时可以确保所有进程有机会执行自身的关闭操作，例如关闭文件、保存当前状态等。然后，向这些进程发送SIGKILL信号，最终彻底终止这些进程。关机操作将试图卸载所有已挂载的文件系统，并调用体系结构专有的关机或重启例程。Linux的shutdown命令与init一起来完成这些操作。

6.7 小结

- Linux系统需要一个根文件系统。不过由于每个应用程序都具有复杂的依赖关系，从零构建根文件系统可能相当困难。
- 文件系统层次结构标准为开发人员部署文件系统并实现最大的兼容性和灵活性提供了参考。
- 我们例举了一个最小文件系统，示范了根文件系统的创建过程。
- Linux内核的最后启动阶段决定并且控制了Linux系统的启动过程。根据嵌入式Linux系统的具体需求，可以采取几种不同的实现方法。
- 详细介绍了init进程，这个功能强大的系统配置和控制工具可以用作嵌入式Linux系统的

基石。本章还介绍了基于init的系统初始化，并例举了几个启动脚本的配置。

- 初始ramdisk是Linux内核的一个特性，它允许用户在内核挂载最终的根文件系统和创建init进程之前进一步定制内核启动过程。本章给出了初始ramdisk的工作机制以及使用这个功能强大的工具的配置举例。
- initramfs简化了初始ramdisk机制，提供了类似的早期引导工具。由于在内核引导时不需要单独加载一个映像文件，同时会在每次内核编译的时候自动对其进行编译，因此initramfs更容易使用。

参考资料

文件系统层次结构标准

由freestandards.org维护

www.pathname.com/fhs/

Boot Process, Init and Shutdown

Linux文档项目

http://tldp.org/LDP/intro-linux/html/sect_04_02.html

Init 帮助手册

Linux文档项目

<http://tldp.org/LDP/sag/html/init.html>

一篇关于System V init的简短描述

<http://docs.kde.org/en/3.3/kdeadmin/ksysv/what-is-sysv-init.html>

Booting Linux: The History and the Future

Werner Almesberger

www.almesberger.net/cv/papers/ols2k-9.ps



本章内容

- 引导装入程序的作用
- 引导装入程序的挑战
- 通用的引导装入程序：Das U-Boot
- 移植U-Boot
- 其他引导装入程序
- 小结

前几章已经提及并列举了引导装入程序的若干操作。引导装入程序（bootloader）是嵌入式系统的一个关键组件，它与其他系统软件的创建提供了基础。本章首先探讨引导装入程序在系统中的作用，接下来介绍引导装入程序所共有的一些特性。有了这些背景知识后，我们将深入研究嵌入式系统中流行的引导装入程序，最后介绍了几个较为流行的引导装入程序。

目前使用的引导装入程序有很多种，想要非常详尽地深入其细节，哪怕是针对最流行的那些引导装入程序，显然也是不切实际的。因此，我们只选择U-Boot来讲解引导装入程序的概念和示例。U-Boot是开源社区中最流行的一种引导装入程序，支持PowerPC、MIPS、ARM和其他一些体系结构。

7.1 引导装入程序的作用

当处理器板加电后，即使运行最简单的程序，也必须对硬件的大量要素进行初始化。每一种体系结构和处理器都有一套预先定义好的动作和配置，包括从板载存储设备（通常是闪存）获取初始化代码。这个早期初始化代码是引导装入程序的一部分，它负责激活处理器和相关的硬件组件。

大多数处理器在加电或复位时会在默认起始地址处获取头几段代码，硬件设计人员则根据这些信息为板载存储设备布线，并选择它要响应的地址范围。这样一来，在系统加电时，代码可以从一个已知的或可预测的地址处获得，从而实现软件控制。

引导装入程序提供了早期初始化代码，并负责初始化主板，以便使其他程序能够运行。这些代码通常由处理器的本机汇编语言编写，这也给我们带来许多挑战，本章将探讨其中一部分。

当然，引导装入程序在完成基本的处理器和平台的初始化后，其主要任务就变成了启动完整的操作系统。它负责定位、载入以及将控制权移交给主操作系统。此外，引导装入程序可能提供一些更高级的特性，比如验证操作系统映像的功能，更新自身或操作系统映像的功能，以及根据开发人员的实现设计在多个操作系统中进行选择的功能。与传统PC机的BIOS模式有所不同，当操作系统取得控制权后，引导装入程序将被覆盖，不复存在^①。

7.2 引导装入程序的挑战

即使是简单的“Hello World” C语言程序也需要相当数量的硬件和软件资源。应用程序开发人员并不需要知道或者过多地关心其中的细节，因为C运行时环境已经悄无声息地提供了这些基础结构。引导装入程序开发人员可没这么幸运，引导装入程序需要的每种资源在使用之前都必须进行详尽的初始化并完成资源分配。其中最明显的例子莫过于DRAM（动态随机访问内存）。

7.2.1 DRAM 控制器

DRAM芯片不能像其他微处理器总线资源那样直接读写，它们需要专门的硬件控制器来支持读写周期。DRAM必须定时刷新，否则其中的数据就会丢失，这会使事情更加错综复杂。刷新操作的实现方式是按照一定规律，并按DRAM厂商规定的时序规范读取DRAM的每个存储单元。目前的DRAM芯片支持多种操作模式，例如针对高性能应用的突发模式和双倍数据速率（DDR）。配置DRAM并保证其按照生产商的时序规范进行刷新，是DRAM控制器的职责，它还负责响应来自处理器的各种读写命令。

设置DRAM控制器是嵌入式开发新手容易受挫的地方。完成这项工作需要详细了解DRAM体系结构、控制器本身、所采用的特定DRAM芯片以及总体的硬件设计。这些内容已经超出本书的范围，对此感兴趣的读者可以阅读本章末尾的参考资源来进一步了解这个重要的概念。附录C介绍了关于这个主题的更多背景知识。

在嵌入式系统中，如果不对DRAM控制器和DRAM进行适当的初始化，基本上什么都做不了。引导装入程序的首要任务就是启用内存子系统。内存完成初始化之后，就可以作为一种资源进行使用。实际上，很多引导装入程序完成内存初始化后的第一个动作，就是将自身复制到DRAM中，以便获得更快的执行速度。

7.2.2 闪存与 RAM

引导装入程序固有的另一个复杂性，是它需要保存在非易失存储器中，但通常又要载入到RAM中才能运行。此外，引导装入程序的复杂性还随它使用的资源增加而增加。在Linux这样完整的操作系统上编译一个程序并从非易失存储器中调用是非常容易的。运行时库、操作系统和编译器一起协作创建了必需的基础设施，负责将程序从非易失存储器中载入到内存，并将控制权移

^① 有些嵌入式设计会保护引导装入程序并为引导装入程序入口函数提供回调函数，但这几乎绝对算不上什么优秀的设计方案。Linux的功能远比引导装入程序强大，因此继续保留引导装入程序往往毫无用处。

交给这个程序。前面提到的“Hello World”程序是一个非常好的例子。在程序被编译后，它可以载入到内存，并通过在命令行上输入程序的名称（hello）执行（当然，这里假定这个可执行文件所在目录已位于PATH变量中）。

在系统加电引导装入程序获得控制权时，上述基础设施并不存在。实际上，引导装入程序必须自行创建一个可操作的上下文环境，并在必要时将自身复制到RAM中的适当位置。另外，还要说明的复杂性是从只读介质中执行程序的需求。

7.2.3 映像的复杂性

作为应用程序开发人员，在为某种熟悉的平台开发程序时，不必关心二进制可执行文件的布局。对于创建包含给定体系结构所需的适当组件的二进制映像，需要预先配置编译器和二进制工具集。链接器会把启动（prologue，序言）和终止（epilogue，尾声）代码加入映像。这些对象构成了应用程序的执行上下文，通常程序开始于main()函数。

但对于普通的引导装入程序而言，却绝对不是这种情况。当引导装入程序获得控制权时，并没有上下文或执行环境。在一个普通的系统中，在引导装入程序初始化处理器和相关硬件之前，很可能并不存在任何DRAM。（请细细品味这句话的含义。）在普通的C函数中，所有局部变量都存储在栈中，因此代码清单7-1中的这个简单函数无法使用。

代码清单7-1 简单的C语言函数

```
int setup_memory_controller(board_info_t *p)
{
    unsigned int *dram_controller_register = p->dc_reg;
    ...
}
```

系统加电后，引导装入程序获得了控制权，此时并没有栈和栈指针。因此，与代码清单7-1相似的C函数很有可能会搞垮处理器，因为编译器会在栈上生成一段代码，用于创建和初始化dram_controller_register指针，而这个指针此时并不存在。引导装入程序必须在调用任何C函数之前创建这个执行环境。

当编译和链接引导装入程序后，开发人员必须非常明白映像是怎样创建并链接的，特别是引导装入程序是如何将自身从闪存搬到RAM的。必须向编译器和链接器传入若干参数，这些参数定义了最终可执行映像特征和布局。有两个主要的特征导致了最终的二进制可执行映像的复杂性。

带来映像复杂性的第一个特征，是需要按照与处理器启动顺序兼容的格式组织启动代码。可执行代码的第一个字节必须根据具体的处理器和硬件体系结构预先定义。例如，AMCC PowerPC 405GP处理器从硬编码地址0xFFFF_FFFC处查找第一条机器指令。其他处理器也会使用类似的方法，不过细节会有些差异。一些处理器配置为系统加电后，根据硬件配置信号的不同，会从几个预先定义好的地址处查找指令代码。

开发人员如何指定二进制映像的布局呢？链接器可以获得一个链接器描述文件，也称作链接器命令脚本。这个特殊文件可以被认为是构建一个二进制可执行映像的诀窍。代码清单7-2取自

一个流行的引导装入程序中链接器描述文件的片段，我们在此简要地讨论。

代码清单7-2 链接器命令脚本

```
SECTIONS
{
    .resetvec 0xFFFF_FFC :
    {
        *(.resetvec)
    } = 0xffff
    ...
}
```

完整地描述链接器命令脚本的语法规则已超出了本书的范围，对此感兴趣的读者可以参考本章末尾给出的GNU LD手册。通过代码清单7-2，可以看到二进制ELF映像文件输出段的定义的开始部分。它指定链接器将调用的.resetvec代码段放置在输出映像的一个固定地址，即0xFFFF_FFC。此外，它还指定了本段剩余的内容应该全用1(0xFFFF)进行填充。这是因为可擦除的闪存阵列包含的都是1。这种技术不仅减少对闪存的磨损，对扇区编程的速度也有提高。

代码清单7-3取自U-Boot最新发行版中定义了.resetvec代码段的文件，它包含在称作.../cpu/ppc4xx/resetvec.S的汇编语言文件中。注意，这个代码段在32位地址的机器中不能超出4字节长度，因为在这个段中只定义了一个指令，而不论定义的配置选项是什么。

代码清单7-3 定义.resetvec的源代码

```
/* Copyright MontaVista Software Incorporated, 2000 */
#include <config.h>
.section .resetvec, "ax"
#if defined(CONFIG_440)
    b _start_440
#else
#if defined(CONFIG_BOOT_PCI) && defined(CONFIG_MIP405)
    b _start_pci
#else
    b _start
#endif
#endif
#endif
```

即使没有汇编语言编程经验，这个汇编文件对你来说也非常容易理解。根据特定的配置（通过CONFIG_*宏指定的），在代码主体的起始位置生成一个无条件分支指令（PowerPC汇编语法中是b指令）。这个分支位置是一个4字节的PowerPC指令，与代码清单7-2中列出的链接器命令脚本片段一样，这个简单的分支指令放置在输出映像的闪存绝对地址0xFFFF_FFC处。正如前面提到的，PPC 405GP处理器从这个硬件解码地址取到它的第一条指令。这就是最初的代码流的定义方式，也是开发人员针对特殊体系结构和处理器的组合所做的必要处理。

7.2.4 执行上下文

引导装入程序映像复杂的另一个主要原因是它缺少执行环境（上下文）。当代码清单7-3中的

指令流开始执行后(回忆一下加电后的第一条机器指令是什么),几乎没有运行程序可用的资源。硬件设计的默认值必须确保从闪存中取到的指令能正常工作,而且系统时钟是有默认值的,不能随便指定^①。每一个处理器的复位状态通常都由生产厂商定义,但是开发板的复位状态由硬件设计人员定义。

的确,大多数处理器在启动时没有可用的DRAM用来临时存储变量,更糟的是,使用C程序的调用需要用到栈,这是一种惯例。如果你非常固执地写一个没有DRAM、没有栈的“Hello World”程序,那么它将与传统的“Hello World”程序极为不同。

这种局限性给设计硬件初始化的原始代码带来了极大的挑战。因此,引导装入程序启动后首要的执行任务之一,就是配置足够数量的RAM以便支持硬件。一些专门为嵌入式系统设计的处理器使用了片上(on-chip)静态RAM。我们讨论的PPC 405GP处理器用的就是这种方法。当RAM可用时,可以使用这部分RAM分配栈空间,同时可以建立适当的上下文运行高级语言,例如C语言。这也使剩下的处理器和平台初始化工作可以使用其他编程语言完成,而不一定是汇编语言。

7.3 通用的引导装入程序: Das U-Boot

有许多开源或商业的引导装入程序可用,而且,很多厂商针对某平台设计的程序现在也在广为应用,其中大多数都具有通用的特性。例如,它们都具备载入和执行其他程序的能力,特别是操作系统。大多数通过串口与用户进行交互。支持不同的网络子系统(例如以太网)不是非常普遍,但却是一个非常有用的特性。

许多引导装入程序都特定于某一种体系结构。对于一些规模较大的开发组织来说,引导装入程序能够支持非常广泛的体系结构和处理器,是非常重要的。对于个体开发,支持多种体系结构的多个处理器也并不少见。为多种交叉平台开发单一的引导装入程序会降低开发成本。

本节将研究在嵌入式Linux社区中已经非常流行的引导装入程序。这个引导装入程序的官方命名是Das U-Boot,由Wolfgang Denk维护,在SourceForge上的主页是<http://u-boot.sourceforge.net/>。U-Boot支持多种体系结构,许多嵌入式开发者和厂商都在项目中采用U-Boot,并从中获益。

7.3.1 执行上下文

要让引导装入程序能够适用于多种处理器和体系结构平台,也需要一些配置手段。就像Linux内核本身那样,引导装入程序的配置在编译时完成。这种方法特别减少了引导装入程序的复杂程度,这也是它的一个重要特性。

在U-Boot中,板级相关的配置由目标平台相关的个别头文件和源码树中的一些软连接指定。这些软连接会根据不同的目标板、体系结构和处理器选择正确的子目录。为支持平台配置U-Boot,可以使用下面的命令:

```
$ make <platform>_config
```

此处platform是U-Boot所支持的众多平台之一。这些平台配置目标在U-Boot顶层目录中的

^① 根据体系结构、处理器和硬件设计的不同,具体细节会有所不同。

makefile文件中列出。例如，为Spectrum Digital OSK开发板配置U-Boot，可以使用下面的命令。该平台使用了TI公司的OMAP 5912处理器：

```
$ make omap5912osk_config
```

这样配置以后，U-Boot源码树将使用恰当的软连接进行选择：ARM作为目标体系结构，ARM926核，5912 OSK作为目标平台。

为该平台配置U-Boot的下一个步骤是编辑与平台相关的配置文件，这个文件在../include/configs子目录中，文件名是omap5912osk.h。随U-Boot发行版提供的README文件中描述了配置的细节内容。README文件也是获得此类信息的最佳资源。

使用配置变量完成U-Boot的配置，这些配置在板级相关的头文件中定义。其中有两种形式的配置变量：使用CONFIG_XXX形式的宏选择配置选项；使用CFG_XXX形式的宏选择配置选项。一般说来，配置选项（CONFIG_XXX）是用户可配置的，它体现了U-Boot的灵活、可选的特征。配置选项（CFG_XXX）通常和硬件相关，需要我们理解处理器和硬件平台的内部细节。与板级相关的U-Boot配置由专门描述具体平台的头文件决定，这个文件包括了针对平台底层的配置选项和设置。这些板级相关的配置头文件都保存在U-Boot源码树的一个子目录中，具体为../include/configs。

在板级相关的配置文件中添加定义，可以选择许多特性和模式。代码清单7-4列出了一个配置文件的部分内容，该文件取自一个假想的基于PPC 405GP处理器的开发板。

代码清单7-4 U-Boot 板级相关配置头文件部分内容

```
#define CONFIG_405GP          /* Processor definition */
#define CONFIG_4XX             /* Sub-arch specification, 4xx family */

#define CONFIG_SYS_CLK_FREQ    33333333 /* PLL Frequency */
#define CONFIG_BAUDRATE        9600
#define CONFIG_PCI              /* Enable support for PCI */
...
#define CONFIG_COMMANDS        (CONFIG_CMD_DFL | CFG_CMD_DHCP)
...
#define CFG_BASE_BAUD          691200

/* The following table includes the supported baudrates */
#define CFG_BAUDRATE_TABLE     \
    {1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400}

#define CFG_LOAD_ADDR          0x100000 /* default load address */
...
/* Memory Bank 0 (Flash Bank 0) initialization */
#define CFG_EBC_PB0AP          0x9B015480
#define CFG_EBC_PB0CR          0xFFF18000

#define CFG_EBC_PB1AP          0x02815480
#define CFG_EBC_PB1CR          0xF0018000
...
```

代码清单7-4给出了U-Boot自身如何配置给定开发板的一种思路，一个实际的板级相关的配置文件可能包含数百行之多。在本例中，你可以看到CPU的定义、CPU系列（4xx）、PLL时钟频率、串口波特率和PCI支持的定义，还包括了配置变量（CONFIG_XXX）和配置设置（CFG_XXX）的实例。最后几行是真实的处理器寄存器的值，初始外部总线控制器需要这些值。你可以发现，这些值只有深入了解开发板和处理器以后才能获得。

通过使用这种机制，可以配置U-Boot的很多方面，包括编译哪些功能（支持DHCP、内存测试、调试等）。这种机制可以用来告知U-Boot，开发板上使用了哪种内存，以及内存大小和内存映射地址。对此感兴趣的读者可以直接阅读U-Boot源代码，特别是非常全面的README文件。

7.3.2 U-Boot 命令集

U-Boot支持多达60个标准的命令集，使用CFG_*宏可以支持超过150个独特的命令。在U-Boot中，通过使用配置设置（CFG_*）宏可以启用命令集。参考附录A就可以查询最近的U-Boot命令的完整列表。这里给出几个命令，通过它们你就可以了解U-Boot的一些功能。

命 令 集	命 令
CFG_CMD_FLASH	闪存命令
CFG_CMD_MEMORY	内存转储、填充、复制、比较等
CFG_CMD_DHCP	DHCP支持
CFG_CMD_PING	Ping命令支持
CFG_CMD_EXT2	EXT2文件系统支持

代码清单7-4中的下面一行定义了U-Boot配置中可用的命令，它在板级相关的头文件中给出：

```
#define CONFIG_COMMANDS (CONFIG_CMD_DFL | CFG_CMD_DHCP)
```

在你自己的板级相关的配置头文件中，如果不想一个一个地输入CFG_*宏，可以从U-Boot源代码中预定义的默认命令集开始。CONFIG_CMD_DFL宏定义了这个默认的命令集，该宏指定了默认的U-Boot命令集列表，例如tftpboot（从tftp服务器启动映像）命令、bootm（从内存启动一个映像）命令、md（显示内存）之类的内存工具，等等。为了启用你自定义的命令组合，需要用默认的方式开头，并根据需要进行添加和删除。代码清单7-4的示例中，增加了DHCP命令。以类似的方式，你可以去除命令：

```
#define CONFIG_COMMANDS (CONFIG_CMD_DFL & ~CFG_CMD_NFS)
```

请查看.../include/configs/中板级相关的头文件获得更多示例。

7.3.3 网络操作

许多引导装入程序都包括了对以太网接口的支持。在开发环境中，这会极大地节省时间。通过串口载入一个中等规模的内核映像可能需要数分钟之久，而使用10Mbit/s的以太网连接仅需几秒钟。此外，使用功能较弱的串口终端的串口连接更容易产生错误。

引导装入程序中一些更重要的特征是包括对BOOTP、DHCP和TFTP协议的支持。这些协议

不常见，BOOTP（启动协议）和DHCP（动态主机控制协议）是允许目标设备通过以太网端口从中央服务器获取IP地址和其他网络相关配置信息的一种协议。TFTP（轻量级文件传输协议）允许目标设备从TFTP服务器下载文件（例如Linux内核映像）。请参考本章结尾列出的这些协议规范。完成这些服务的服务器将在第12章加以说明。

图7-1阐述了目标设备和BOOTP服务器之间的信息流。客户（在本案例中是U-Boot）开始发送一个广播包搜索BOOTP服务器。这个服务器返回一个数据包作为响应，该数据包中包含了客户端的IP地址和其他信息。最有用的数据是下载的内核映像的文件名。

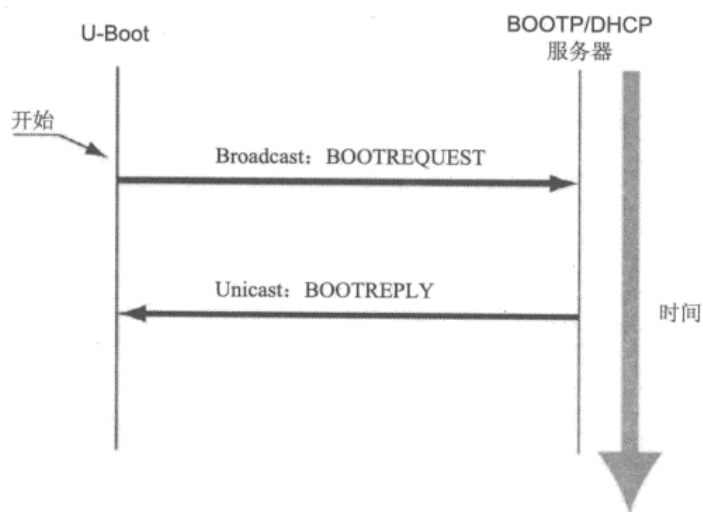


图7-1 BOOTP客户/服务器握手

在开发过程中，BOOTP服务器不再作为单一的服务器存在。在你喜欢的Linux发行版中包括了DHCP服务器，也同样包括了BOOTP协议。

DHCP协议建立在BOOTP之上，它能为目标平台提供相当多的配置信息。在开发过程中，信息交换通常被目标/引导装入程序（DHCP客户端）限制。代码清单7-5列出了一个DHCP服务器的配置，该配置可以识别一个目标设备。该片段取自Fedora Core 2中DHCP服务的配置文件。

代码清单7-5 DHCP目标规范

```

host coyote {
    hardware ethernet 00:0e:0c:00:82:f8;
    netmask 255.255.255.0;
    fixed-address 192.168.1.21;
    server-name 192.168.1.9;
    filename "coyote-zImage";
    option root-path "/home/chris/sandbox/coyote-target";
}
...

```

当DHCP服务器收到来自与硬件以太网地址（代码清单7-5中给出）相匹配的设备的数据包时，

它将把参数按照目标规范的格式发送给目标设备来作出响应。表7-1说明了目标规范中的字段。

表7-1 DHCP目标参数

DHCP目标参数	用 途	注 释
host	主机名	DHCP 配置文件的符号标签
hardware ethernet	以太网硬件地址	目标机的以太网接口使用的底层以太网硬件地址
fixed-address	目标机IP地址	目标机使用的IP地址
netmask	目标机子网掩码	目标机使用的IP子网掩码
server-name	TFTP 服务器 IP 地址	IP地址, 目标机以此地址直接请求文件传输、根文件系统等
filename	TFTP下载的文件名	引导装入程序使用该文件名启动第2个映像 (一般来说是Linux内核)

当目标板上的引导装入程序完成BOOTP或DHCP交换以后, 前面说过的参数将用于更多的配置。例如, 引导装入程序会使用目标板的IP地址和它的以太网端口进行绑定, 然后使用server-name列指定的目的IP地址请求filename列中的文件, 在大多数情况下, 这个文件通常是Linux内核映像。虽然这是大多数情况, 但也可以想象它可能会是厂商测试和诊断固件的程序。

请注意, DHCP协议支持的参数要远远多于表7-1列出的, 表中给出的只是你在嵌入式系统中最有可能遇到的一些参数。要获得完整的信息请参考本章结尾给出的DHCP规范参考资料。

7.3.4 存储子系统

很多引导装入程序具有支持从各种存储设备而不仅仅是常用的闪存中启动映像的能力。支持多种存储类型设备的难度在于软硬件之间的复杂联系。例如, 要访问硬盘上的数据, 引导装入程序必须有IDE控制器接口的驱动程序, 还要了解分区方案和文件系统类型。这并非没有价值, 它是成熟操作系统中的一项任务。

虽然有些复杂, 但从这类设备中载入映像还是有多种方法可寻的。最简单的方法是只支持硬件。使用这种方法, 不需要了解文件系统的知识, 引导装入程序只是从设备的绝对扇区里载入数据。这种方法适用在与IDE兼容设备 (例如CF卡) 的未格式化的分区中, 从0扇区开始载入找到的数据, 这些数据没有任何的结构组织。对于载入内核映像或者其他二进制映像文件, 这是一种理想的配置。设备上的其他分区可以被格式化为指定的文件系统。内核启动以后, 设备驱动程序用来访问其他分区。

U-Boot可以从指定的分区或带有文件系统结构的分区载入映像, 当然, 开发板必须支持硬件设备 (IDE子系统), 同时U-Boot也必须按此进行配置。在板级相关的配置文件中增加CFG_CMD_IDE后, 就启用了对于IDE接口的支持, 且增加CFG_CMD_BOOTD将启用从原始分区启动的功能。如果要将U-Boot移植到自己的开发板, 你就必须修改U-Boot代码以满足特殊的硬件。

7.3.5 从磁盘启动: U-Boot

正如前一节所述, U-Boot支持几种从磁盘子系统启动内核映像的方法。下面这条简单的命令说明了其中支持的一种方法:

```
=> diskboot 0x400000 0:0
```

要了解上面的语法，必须首先理解U-Boot是如何计数磁盘设备的。本例中的0:0说明了设备和分区。在这个简单的实例中，U-Boot执行一个原始的二进制程序，这个程序是从所找到的第1块IDE设备（IDE设备0）的第1个分区中载入的映像，该映像会被载入到物理地址0x400000处。

内核映像被载入到内存后，U-Boot使用bootm（从内存启动）命令启动内核：

```
=> bootm 0x400000
```

7.4 移植 U-Boot

U-Boot变得如此流行，原因之一是它很容易支持新的平台。移植新开发板，必须提供一个下级的makefile文件，该文件提供了构建过程中用到的板级相关的定义。这些makefile文件都以config.mk命名，并保存在U-Boot顶层源文件目录下的.../board/xxx子目录中，其中xxx指的是一个特殊的开发板。

在最新的U-Boot 1.1.4版本中，.../boards子目录下有240多个不同的开发板配置文件，它们都命名为config.mk。在这个版本中，支持29个不同的CPU配置（按同样方式计数后得出）。注意在一些情况下，一种CPU配置会涉及一个系列的芯片，例如ppc4xx就支持了PowerPC 4xx系列的几款处理器。目前，U-Boot支持了很多目前正在使用的流行的处理器和处理器系列，基于这些处理器的参考板也得到了U-Boot的支持。

如果开发板上的CPU是所支持的CPU之一，那么移植U-Boot就非常简单了。如果必须增加新的CPU，那么就需要下一番苦功了。不过有幸的是，也许有人已经在你之前完成了大部分工作。不管是基于现有的CPU移植新的CPU还是移植新的开发板，都应该仔细研究与之相对应的源代码，确定最贴近你使用的那款CPU，复制CPU相关目录下的功能函数。最后，修改源代码，增加对新CPU需求的特殊支持。

7.4.1 为 EP405 开发板移植 U-Boot

将U-Boot移植到一个新平台会使用相同的方法，下面举例说明。我们将要使用的开发板叫作EP405（Embedded Planet），板上使用了AMCC PowerPC 405GP处理器。本例中使用的这块特殊的开发板上有64MB的SDRAM和16MB的Flash，以及其他一些设备。

第一步是看一下U-Boot中已经支持的开发板与我们使用的开发板有多少差距。U-Boot源码树中，有很多支持405GP处理器的开发板，使用grep快速查找对开发板进行配置的头文件：

```
$ cd .../u-boot/include/configs$ grep -l CONFIG_405GP *
```

在最新的U-Boot中，针对405GP处理器进行配置的文件有25个。在查看几个以后，选择AR405.h文件作为我们移植工作的基线，文件中提供了对LXT971以太网的支持。我们的目标就是借鉴开源，尽量减少我们的开发工作量。先完成简单的步骤：复制一份开发板的配置文件，这个新文件的文件名是根据你的开发板自行设置的，这里叫EP405.h。在U-Boot源码树的顶层目录执行下面命令，完成上述步骤：

```
$ cp ../include/configs/AR405.h ../include/configs/EP405.h
```

接下来创建与开发板相关的目录，将AR405开发板的文件复制到该目录中。此时，我们还不是否需要其中的全部文件，这一步会在后面讲到。复制这些文件后，修改文件名，以匹配自己的开发板。

```
$ cd board <<< from top level U-Boot source directory
$ mkdir ep405
$ cp esd/ar405/* ep405
```

最难的一步到来了。Jerry Van Baren是对U-Boot有贡献的开发者，在U-Boot邮件列表中，他用诙谐幽默的语言详细说明了移植U-Boot的过程。他所提供的完整移植过程可以在U-Boot的README文件中找到，是用C语言写的。下面使用Jerry的风格来总结移植过程的难点部分：

```
while (!running) {
    do {
        Add / modify source code
    } until (compiles);
    Debug;
    ...
}
```

正如这里总结的，Jerry的移植过程简单而又正确。当你确定了基线后，必须增加、删除和修改源代码，直到编译，然后在正常运行之前进行调试。这个过程没有捷径。将任何引导装入程序移植到一个新的开发板都需要你掌握很多领域的知识，包括软件和硬件。其中一些知识点相当专业和复杂，例如SDRAM控制器。事实上，所有这些工作都包括具备硬件底层的细节知识。忠告：把大量的娱乐时间用到阅读处理器硬件参考手册上以及板上其他组件的参考手册。

7.4.2 U-Boot 的 makefile 配置目标

既然我们是从已有的代码开始的，那么必须在U-Boot顶层目录的makefile中做些修改，在里面增加针对我们自己开发板的配置。在查看这个makefile以后，我们可以找到一段针对所支持的不同开发板配置U-Boot源代码的内容，在这里增加我们的新开发板以进行编译。因为我们的开发板是从ESD AR405衍生的，因此要将其规则作为一个模板，用它进行编译。如果你继续往下阅读U-Boot的源代码，会看到这些规则在makefile中是用配置名按照字母顺序排列的。我们是开源世界的公民，理应遵循其规则。与U-Boot的约定一致，我们称这个配置为EP405_config。

```
EBONY_config:      unconfig
                  @./mkconfig $(@:_config=) ppc ppc4xx ebony

+EP405_config:      unconfig
+                  @./mkconfig $(@:_config=) ppc ppc4xx ep405
+
ERIC_config:        unconfig
                  @./mkconfig $(@:_config=) ppc ppc4xx eric
```

我们的新配置规则已经插入其中，即前面加有+字符的三行（标准的diff格式）。

完成了这些步骤后，就有了代表起点的U-Boot源码树。它可能不会被无误地编译，但这是我

们的第一步。至少编译过程可以给一些提示，以便我们知道从哪里入手。

7.4.3 EP405 处理器初始化

新移植的U-Boot的第一个必须正确完成的任务，是初始化处理器和内存（DRAM）子系统。复位后，405GP处理器核设计为从0xFFFF_FFFC地址处开始取指。处理器核尝试执行此地址处的指令。因为这是内存的最高端，所以这里的指令必须是非条件分支指令。

这个处理器核也是通过硬编码配置了最高2MB的内存空间的，从而即使不对外部总线控制器进行编程也可以访问到它，这段空间通常供闪存设备使用。这迫使分支指令必须落在这段地址空间里，因为处理器不能编码，除非引导装入程序初始化了额外的内存空间。我们必须跳转到在0xFFE0_0000地址附近或之上。我们是怎么知道这些的呢？因为我们阅读了405GP用户手册。

正如前面描述，405GP处理器核启动的必要条件通过硬件设计完成，以确保加电后非易失内存（Flash）能被映射到所需要的2MB内存空间里。这段初始内存空间的某些属性假定了复位时的默认值。例如，这个最高2MB的空间将被配置为256个等待状态、3个芯片地址选择延迟周期、3个片选输出使能延迟周期以及7个保持周期^①。这样，硬件设计工程师在选择适当设备时，或在系统复位后想直接获得处理器执行的指令代码时，就能获得最大程度的自主权。

在代码清单7-2中，我们已经看到了如何在Flash的最上面安装复位向量表。.../cpu/ppc4xx/start.S文件中的前几行代码是为405GP处理器核进行配置的。U-Boot开发人员有意地把这段代码写成和处理器无关的。理论上，这个文件中不需要板级相关的代码。你可以看看该文件是怎么完成的。

不论你对start.S中的逻辑流理解到什么程度，都不需要理解PowerPC汇编语言。关于修改底层汇编代码的内容，在U-Boot邮件列表的很多常见问题（FAQ）中已经给出了答案。在几乎所有的情况下，如果U-Boot移植到所支持的处理器上，都不需要修改这段代码。这是成熟的代码，很多成功案例移植的都是这段程序。移植时，你只需要修改板级相关的代码（最少）。如果你发现自己身陷困境，或是在处理器早期启动的汇编代码处遇到麻烦，极有可能是你的方向不正确。

代码清单7-6给出了4xx体系结构的start.S文件的一部分。

代码清单7-6 U-Boot中4xx的启动代码

```
...
#if defined(CONFIG_405GP) || defined(CONFIG_405CR) ||
    defined(CONFIG_405) || defined(CONFIG_405EP)
    /*----- */
    /* Clear and set up some registers. */
    /*----- */
    addi    r4,r0,0x0000
    mtspr   sgr,r4
    mtspr   dcwr,r4
    mtesr   r4                /* clear Exception Syndrome Reg */
    mttcr   r4                /* clear Timer Control Reg */
```

① 数据直接取自405GP的用户手册，请参考本章后面的内容。


```

mtxer  r4          /* clear Fixed-Point Exception Reg */
mtevpr r4          /* clear Exception Vector Prefix Reg */
addi   r4,r0,0x1000 /* set ME bit (Machine Exceptions) */
oris   r4,r4,0x0002 /* set CE bit (Critical Exceptions) */
mtmsr  r4          /* change MSR */
addi   r4,r0,(0xFFFF-0x10000) /* set r4 to 0xFFFFFFFF (status in the */
                                   /* dbcsr is cleared by setting bits to 1) */
mtdbsr r4          /* clear/reset the dbcsr */

/*----- */
/* Invalidate I and D caches. Enable I cache for defined memory regions */
/* to speed things up. Leave the D cache disabled for now. It will be */
/* enabled/left disabled later based on user selected menu options. */
/* Be aware that the I cache may be disabled later based on the menu */
/* options as well. See miscLib/main.c. */
/*----- */
bl      invalidate_icache
bl      invalidate_dcache

/*----- */
/* Enable two 128MB cachable regions. */
/*----- */
addis   r4,r0,0x8000
addi    r4,r4,0x0001
mticcr  r4          /* instruction cache */
isync

addis   r4,r0,0x0000
addi    r4,r4,0x0000
mtdccr  r4          /* data cache */

```

405GP处理器的start.S文件中，第一条执行的代码大约在文件的三分之一处，这里将清除或设置很多处理器的默认值。接着禁用指令和数据缓存，开启指令缓存以加速初始的加载过程。建立了两块128MB可缓存的区域：一个在高端内存（闪存区域），另一个在底部（一般是在系统DRAM的开始处）。最后，U-Boot被复制到这个区域的RAM中，并在那里执行。这样做是出于性能方面的考虑：从RAM读取数据的速度比从闪存中读取要快上几个数量级（甚至更快）。但是，对于4xx处理器来说，开启指令缓存有另一个巧妙的原因，我们很快揭晓其中的奥妙。

7.4.4 特定开发板的初始化

特定板初始化第一个时机是在.../cpu/ppc4xx/start.S文件中缓存区被初始化之后。这里我们可以找到一个叫作ext_bus_cntlr_init的外部汇编语言例程。

```
bl ext_bus_cntlr_init /* Board specific bus cntlr init */
```

这个例程在.../board/ep405/init.S文件中定义，init.S文件在特定开发板目录中。它提供一个针对非常早期的硬件初始化的钩子，是我们为EP405平台定制的文件之一。这个文件包含板级相关的代码，用来初始化405GP的外部总线控制器。代码清单7-7给出文件大体功能的实体部分。这就是初始化405GP的外部总线控制器的代码。

代码清单7-7 外部总线控制器初始化

```

        .globl ext_bus_cntlr_init
ext_bus_cntlr_init:
    mflr    r4          /* save link register      */
    bl      ..getAddr
..getAddr:
    mflr    r3          /* get _this_ address      */
    mtlr    r4          /* restore link register   */
    addi    r4,0,14     /* prefetch 14 cache lines */
    mtctr   r4          /* ...to fit this function */
                /* cache (8x14=112 instr) */

..ebcloop:
    icbt    r0,r3       /* prefetch cache line for [r3] */
    addi    r3,r3,32    /* move to next cache line */
    bdnz    ..ebcloop  /* continue for 14 cache lines */

    /*-----*/
    /* Delay to ensure all accesses to ROM are complete */
    /* before changing bank 0 timings */
    /* 200usec should be enough. */
    /* 200,000,000 (cycles/sec) X .000200 (sec) = */
    /* 0x9C40 cycles */
    /*-----*/

    addis   r3,0,0x0
    ori     r3,r3,0xA000 /* ensure 200usec have passed t */
    mtctr   r3

..spinlp:
    bdnz    ..spinlp    /* spin loop */

    /*-----*/
    /* Now do the real work of this function */
    /* Memory Bank 0 (Flash and SRAM) initialization */
    /*-----*/

    addi    r4,0,pb0ap   /* *ebccfga = pb0ap; */
    mtdcr   ebccfga,r4
    addis   r4,0,EBC0_B0AP@h /* *ebccfgd = EBC0_B0AP; */
    ori     r4,r4,EBC0_B0AP@l
    mtdcr   ebccfgd,r4

    addi    r4,0,pb0cr   /* *ebccfga = pb0cr; */
    mtdcr   ebccfga,r4
    addis   r4,0,EBC0_B0CR@h /* *ebccfgd = EBC0_B0CR; */
    ori     r4,r4,EBC0_B0CR@l
    mtdcr   ebccfgd,r4

    /*-----*/
    /* Memory Bank 4 (NVRAM & BCSR) initialization */
    /*-----*/

    addi    r4,0,pb4ap   /* *ebccfga = pb4ap; */

```

```

mtdcr    ebccfga,r4
addis    r4,0,EBC0_B4AP@h /* *ebccfgd = EBC0_B4AP; */
ori      r4,r4,EBC0_B4AP@l
mtdcr    ebccfgd,r4

addi     r4,0,pb4cr        /* *ebccfga = pb4cr; */
mtdcr    ebccfga,r4
addis    r4,0,EBC0_B4CR@h /* *ebccfgd = EBC0_B4CR; */
ori      r4,r4,EBC0_B4CR@l
mtdcr    ebccfgd,r4

blr                               /* return */

```

选择代码清单7-7中的示例是因为在底层处理器初始化中，这段代码采用了非常巧妙的技术。对于认识代码运行的上下文来说，这段代码很重要。它从闪存中执行，此时没有DRAM可用，没有栈，这段代码准备对控制器做一些基本改变，从而获得控制此刻正在执行的闪存的访问权。这在处理器文档中写得很清楚，如果在修改外部总线控制器的同时从所附着的闪存中执行代码，会导致数据读取错误和处理器的崩溃。

解决方案在这个示例汇编语言程序中。从`..getAddr`标签处开始到后面7条汇编语言指令，代码的含义是，使用`icbt`指令将自身预取指到指令缓存中。在整个子程序成功地读入指令缓存以后，它将继续对外部总线控制器做一些需要的改动，而无需担心崩溃，这是因为它直接从内部指令缓存中执行。微妙，但又聪明！这段代码后面是一个短暂的延时，以确保i-cache能读到全部请求。

完成预取指和延时以后，代码接下来要做的是根据开发板配置第0组和第4组内存，具体的值根据板上组件的详细内容以及它们的连接方式而定。对本例提到的PowerPC汇编语言和405GP处理器感兴趣的读者可以参考本章最后的“参考资源”。

如果没有完全明白这段代码的含义，请不要考虑改动它。也许你增加了几行，致使代码容量超出缓存范围，那么系统很可能会崩溃（更糟的是它可能只在某些时候崩溃），而且如果用调试器单步跟踪这段代码，也还是无法找出问题所在。

特定板初始化的下一个时机是在处理器数据缓存中分配临时栈之后。这段内容在`.../cpu/ppc4xx/start.S`文件的第727行、初始化SDRAM控制器的分支中：

```
bl sdrain_init
```

执行上下文现在包括一个栈指针和一些用于存储本地数据的临时内存，即部分C上下文，它使得开发人员可以用C语言完成系统SDRAM控制器设置和其他初始化等一些相对复杂的任务。在移植EP405时，`sdrain_init()`函数在`.../board/ep405/ep405.c`文件中，该函数针对这个特定的开发板进行了定制和DRAM的配置。因为这块开发板没有使用商业上可用的DRAM内存SIMM（Single Inline Memory Module），所以它不可能动态配置DRAM。与U-Boot支持的很多其他开发板一样，这个过程在`sdrain_init`函数中完成。

很多现成的DDR模块使用SPD（Serial Presence Detect，串行存在性检测）PROM保存定义内存模块的参数。这些参数可以通过程序的控制读取，一般使用I2C总线。这些参数还可用来作为

输入，以帮助内存控制器决定合适的参数。U-Boot支持这种技术，但是它可能需要根据特定开发板做些修改。在U-Boot源代码中有很多这样的实例。CONFIG_SPD_EEPROM配置选择用来开启这个特性。你可以搜索这个选择以找到使用它的例子。

7.4.5 移植概要

到目前为止，你已经了解了一些将引导装入程序移植到硬件平台的困难了。除了要深入掌握硬件知识以外，这也没有什么难的。当然，我们最好在任务期内花最少的时间在这上面。毕竟我们平时没有把精力放在理解处理器的每一处细节上，而更习惯于及时地拿出项目解决方案，以展示我们的能力。实际上，这也正是开源精神长盛不衰的主要原因之一。我们刚才说将U-Boot移植到新平台是多么的简单，不是因为我们是处理器领域的世界级专家，而是因为我们之前的很多人已经完成了大量困难的工作。

代码清单7-8是一份完整的文件列表，里面记录了将U-Boot移植到EP405时新增或修改的文件。当然，如果有U-Boot中不支持的新硬件设备，或者我们移植了一个U-Boot还不曾支持的处理器，那么所做的工作将要多得多。虽然听起来可能多余，但这里仍要指出的是，在合理的期限内，成功移植的关键是完全掌握硬件（处理器和子系统）和软件（U-Boot）的细节知识点，此外别无他法。如果你带着这种心态开始项目，你一定会成功！

代码清单7-8 将U-Boot移植到EP405时新增或修改的文件

```
$ diff -purN u-boot u-boot-ep405/ | grep +++
+++ u-boot-ep405/board/ep405/config.mk
+++ u-boot-ep405/board/ep405/ep405.c
+++ u-boot-ep405/board/ep405/ep405.h
+++ u-boot-ep405/board/ep405/flash.c
+++ u-boot-ep405/board/ep405/init.S
+++ u-boot-ep405/board/ep405/Makefile
+++ u-boot-ep405/board/ep405/u-boot.lds
+++ u-boot-ep405/include/config.h
+++ u-boot-ep405/include/config.mk
+++ u-boot-ep405/include/configs/EP405.h
+++ u-boot-ep405/include/ppc405.h
+++ u-boot-ep405/Makefile
```

回忆.../board/ep405目录中的文件，这些文件源于另一个目录。实际上，我们在本次移植中，并不需要从头开始创建任何文件。我们借鉴了他人的工作成果，根据我们的目标做了必要的定制。

7.4.6 U-Boot 映像格式

我们已经有了一个能运行在EP405开发板上的引导装入程序，现在就可以利用它载入并运行程序。理想情况下，我们希望运行操作系统，比如Linux。因此，我们需要了解U-Boot所需的映像格式。U-Boot在映像文件前面添加一段首部，以和其他映像区别开。U-Boot通过mkimage工具（U-Boot源码的一部分）创建这个映像首部。

最新的Linux内核发行版已经内建支持直接构建U-Boot可启动的映像。内核源码树中，ARM和PPC分支都支持名为uImage的映像。我们看看PPC中的uImage。下面的代码取自Linux内核中PPC的makefile，它的位置在.../arch/ppc/boot/images/Makefile。该文件提供了构建uImage映像的规则：

```
quiet_cmd_uimage = UIMAGE $@
cmd_uimage = $(CONFIG_SHELL) $(MKIMAGE) -A ppc \
-O linux -T kernel -C gzip -a 00000000 -e 00000000 \
-n 'Linux-$(KERNELRELEASE)' -d $< $@
```

不用理会复杂的语法，我们需要明白\$(MKIMAGE)变量的含义。shell脚本将执行U-Boot的mkimage工具，并使用你看到的一些参数。mkimage工具创建U-Boot首部，并加在Linux内核映像之上。参数定义如下：

- -A，指定目标映像的体系结构；
- -O，指定目标映像的操作系统，本例中是Linux；
- -T，指定目标映像的类型，本例中是内核；
- -C，指定目标映像的压缩类型，这里是gzip；
- -a，设置U-Boot的加载地址（loadaddress），本例中是0；
- -e，设置U-Boot映像的入口点（entry point）地址；
- -n，一段用来给用户识别映像的文本信息；
- -d，信息首部加载的可执行映像文件名称。

有几个U-Boot命令使用这个首部的数据校验映像完整性（U-Boot会在首部中放置一个CRC签名），还利用这些数据指示各种命令对这个映像的处理。U-Boot有一个命令叫imininfo，它会读取映像的首部，并从目标映像中显示属性。代码清单7-9包含了通过U-Boot的tftpbboot命令载入uImage（U-Boot格式的Linux内核映像）并在映像上执行imininfo命令的结果。

代码清单7-9 U-Boot的imininfo命令

```
=> tftpbboot 400000 uImage-ep405
ENET Speed is 100 Mbps - FULL duplex connection
TFTP from server 192.168.1.9; our IP address is 192.168.1.33
Filename 'uImage-ep405'.
Load address: 0x400000
Loading: ##### done
Bytes transferred = 891228 (d995c hex)
=> imininfo

## Checking Image at 00400000 ...
Image Name: Linux-2.6.11.6
Image Type: PowerPC Linux Kernel Image (gzip compressed)
Data Size: 891164 Bytes = 870.3 kB
Load Address: 00000000
Entry Point: 00000000
Verifying Checksum ... OK
=>
```

7.5 其他引导装入程序

这里要介绍其他一些流行的引导装入程序，说说它们可能在哪里使用，并总结它们的特性。我们并不打算给出一份完整的说明，因为这样做将需要一本书的篇幅。对此感兴趣的读者可以参考本章后面的“参考资源”，做进一步的学习。

7.5.1 Lilo

Linux加载器（即Lilo）在桌面PC平台的商业Linux发行版中广泛使用。同样，在Intel x86/IA32体系结构中也使用得很多。Lilo由几个组件组成。它在可启动磁盘的第1个扇区上有一个主加载器^①。这个主加载器限制在磁盘扇区容量以内，通常是512字节。因此，Lilo的主要任务是直接载入二级加载器，并将控制权移交给它。二级加载器可以跨越多个分区，它将完成加载器的大多数工作。

Lilo由一个配置文件和工具驱动，这个工具是lilo可执行程序中的一部分。配置文件在主机操作系统的控制下进行读写，也就是说，配置文件既不被主加载器引用，也不被二级加载器引用。配置文件的条目是在系统安装或系统管理时，通过lilo配置工具读取或处理的。代码清单7-10是一个简单的lilo.conf配置文件的示例，文件中描述了典型的Linux和Windows双系统启动。

代码清单7-10 Lilo 配置文件：lilo.conf

```
# This is the global lilo configuration section
# These settings apply to all the "image" sections

boot = /dev/hda
timeout=50
default=linux
# This describes the primary kernel boot image
# Lilo will display it with the label 'linux'
image=/boot/myLinux-2.6.11.1
    label=linux
    initrd=/boot/myInitrd-2.6.11.1.img
    read-only
    append="root=LABEL=/"

# This is the second OS in a dual-boot configuration
# This entry will boot a secondary image from /dev/hda1
other=/dev/hda1
    optional
    label=that_other_os
```

这个配置文件指示Lilo配置工具使用第1块硬盘（/dev/hda）的主引导记录（master boot record, MBR）。文件还包含了一条用户延迟指令，给用户一个在超时以前进行选择的机会（本例中是50s）。这也给系统使用者提供了从列表中选择启动哪一个操作系统映像的机会。如果操作

^① 这主要是历史原因：从早期的PC时代，BIOS程序都只从磁盘的第1个扇区载入，并传递控制权。

系统使用者在时间用尽以前按下Tab键，Lilo将列出一张列表以供选择。Lilo使用label标签显示每一个映像文件。

配置文件中的映像通过image标签定义。见代码清单7-10，本例中的主要（默认）映像是一个Linux内核映像，文件名是myLinux-2.6.11.1。Lilo将从硬盘载入这个映像。之后，它将载入用作初始化ramdisk的第2个文件，即myInitrd-2.6.11.1.img。Lilo用包含"root=LABEL=/"的字符串构建内核命令行，并在Linux内核执行过程中传递给内核。Linux在启动后，将通过这个方法获得根文件系统。

7.5.2 GRUB

现在有很多商业Linux发行版都使用GRUB作为引导装入程序。GRUB（Grand Unified Bootloader）是一个GNU项目，提供了许多Lilo所没有的特性。GRUB和Lilo之间最大的不同是，GRUB能够理解文件系统和内核映像格式。而且，GRUB可以在启动时读取和修改其配置选项。GRUB也支持通过网络启动，这在嵌入式环境中极为有用。GRUB在启动时会提供一个命令行接口，利用此接口，就可以在启动过程中修改配置参数。

与Lilo一样，GRUB也通过一个配置文件进行驱动。但是和Lilo的静态配置有些不一样，GRUB引导装入程序在启动时读取这个配置。这意味着可以根据不同的系统配置，在启动过程中进行修改。

代码清单7-11给出一个GRUB配置文件的示例，这个配置文件取自PC机。GRUB的配置文件名为grub.conf，通常存放在一个专门存储启动映像文件的小分区中。在本例的配置文件所在的机器里，这个目录是/boot。

代码清单7-11 GRUB 配置文件示例：grub.conf

```
default=0
timeout=3
splashimage=(hd0,1)/grub/splash.xpm.gz

title Fedora Core 2 (2.6.9)
    root (hd0,1)
    kernel /bzImage-2.6.9 ro root=LABEL=/ rhgb proto=imps quiet
    initrd /initrd-2.6.9.img

title Fedora Core (2.6.5-1.358)
    root (hd0,1)
    kernel /vmlinuz-2.6.5-1.358 ro root=LABEL=/ rhgb quiet

title That Other OS
    rootnoverify (hd0,0)
    chainloader +1
```

GRUB首先给用户列出可启动的映像。代码清单7-11中的title是用户可以看到的映像名。default标签指定在给定时间内（本例为3s）不进行选择将自动启动的映像。映像的序号从0开始计数。

与Lilo不同，GRUB可以读取一个给定分区上的文件系统，并载入映像。root标签符指定根分区，grub.conf配置文件中的所有文件名都在根分区中。在本例所示的配置中，根是第1块磁盘的第1个分区，用root(hd0,1)表示。分区从0开始计数，这里指的是第1块磁盘的第2个分区。

映像被指定为与所指定的根分区相关的文件名。在代码清单7-11中，默认启动的映像是Linux 2.6.9内核，与之相匹配的ramdisk映像是initrd-2.6.9.img。注意，GRUB的语法是将内核命令行参数与内核文件位置放在同一行上。

7.5.3 其他引导装入程序

很多其他的引导装入程序也用在专用的环境中。例如，Redboot是另一个用于Intel和Xscale平台上的开源引导装入程序，它用于采用Intel IXP和PXA系列处理器的评估板中；Micromonitor用在诸如Cogent和其他一些厂商的开发板中；YAMON一般用于MIPs平台中^①；LinuxBIOS主要用在X86环境中。总之，当你考虑选择一个启动代码时，应该首先考虑如下一些重要的因素：

- 它支持我选择的处理器吗？
- 它是否已经移植到了与我使用的开发板相似的板子上？
- 它支持我所需要的特性吗？
- 它支持我计划使用的硬件设备吗？
- 我是否能够获得社区里的很多用户的支持？
- 有没有我可以购买支持服务的商业厂商？

7.6 小结

- 引导装入程序在嵌入式系统的作用怎么强调也不为过，它是系统加电后运行的第一个软件。
- 本章解释了引导装入程序的作用，并探讨了引导装入程序必须面对的受限执行上下文。
- Das U-Boot成为众多处理器体系结构中一个很流行的通用引导装入程序，它支持大量处理器、参考硬件平台和定制的开发板。
- U-Boot使用一个特定板级的头文件中的一系列配置变量进行配置。附录A包含了U-Boot最新发布所支持的全部标准命令集列表。
- 如果新平台使用的处理器已经被U-Boot支持，那么将U-Boot移植到这个新平台是相当简单的。在本章中，我们简略介绍了将U-Boot移植到新平台的一般步骤。
- 要完成引导装入程序的移植或修改，你必须掌握所使用的处理器和硬件平台的详细知识。
- 我们简要介绍了目前使用的其他引导装入程序，这样你可以根据自身需要进行选择。

参考资源

Application Note: Introduction to Synchronous DRAM

^① 在现在存在的很多引导装入程序中，YAMON的用户指南称自己是“Yet Another MONitor”。

Maxwell Technologies

www.maxwell.com/pdf/me/app_notes/Intro_to_SDRAM.pdf

Using LD, the GNU linker

自由软件联盟

www.gnu.org/software/binutils/manual/ld-2.9.1/ld.html

The DENX U-Boot and Linux Guide (DLUG) for TQM8xxL

Wolfgang Denx et al., Denx Software Engineering

www.denx.de/twiki/bin/view/DULG/Manual

RFC 793, "Trivial File Transfer Protocol"

The Internet Engineering Task Force

www.ietf.org/rfc/rfc793.txt

RFC 951, "Bootstrap Protocol"

The Internet Engineering Task Force

www.ietf.org/rfc/rfc951.txt

RFC 1531, "Dynamic Host Control Protocol"

The Internet Engineering Task Force

www.ietf.org/rfc/rfc1531.txt

PowerPC 405GP 嵌入式处理器用户手册

International Business Machines, Inc.

32位PowerPC体系结构编程环境手册

Freescale Semiconductor, Inc.

Lilo 引导装入程序

www.tldp.org/HOWTO/LILO.html

GRUB 引导装入程序

www.gnu.org/software/grub/



本章内容

- 设备驱动程序基本概念
- 模块实用程序
- 驱动程序方法
- 汇总
- 设备驱动程序与GPL
- 小结

系统设计中更有挑战性的工作之一就是合理地划分系统的每一项功能。由UNIX或者Linux操作系统提供的设备驱动程序模型在用户的应用程序代码和系统硬件或者内核设备之间进行了自然的切分。本章将讲解设备驱动程序模型以及Linux设备驱动程序体系结构的基础知识。学习本章后，你就打下了坚实的基础，可以借助本章最后介绍的那些优秀文章，继续深入研究设备驱动程序。

本章将首先介绍Linux设备驱动程序的一些基本概念，以及内核源码树中设备驱动程序的构建系统。我们会仔细讲解Linux设备驱动程序的体系结构，并且动手编写一个简单的、可以工作的驱动程序实例。本章还要介绍一些常用的实用工具，这些工具用来在用户层实现内核设备驱动程序模块的加载与卸载工作^①，并通过一个简单的应用程序来解释应用程序与设备驱动程序之间的接口；最后还会介绍设备驱动程序与GNU公共许可（GPL）之间的关系。

8.1 设备驱动程序基本概念

很多有经验的嵌入式系统开发人员一开始都会觉得虚拟内存操作系统中设备驱动程序的概念有些难度。这是因为目前很多遗留的实时操作系统的体系结构都与此不同。虚拟内存和内核工作空间/用户空间等新概念的引入常常增加了复杂性。

设备驱动程序的一个最基本的目的，就是将用户应用程序与关键的内核数据结构或者系统硬件设备分割开，一个编写良好的设备驱动程序应该能够有效地对用户隐藏硬件设备的复杂性以及

^① 在本章中，设备驱动程序和模块是同一概念。

可变性。例如，一个程序需要向硬盘写入数据，那么这个程序不需要关心当前的硬盘扇区究竟是512字节还是1024字节，用户只要简单地打开相应的文件然后发出写入数据的命令就可以了，那些底层的细节都由设备驱动程序来处理，这样就将用户与复杂而又危险的硬件设备编程分割开了。设备驱动程序提供了一个面向大量不同硬件设备的一致性用户接口。在UNIX/Linux操作系统中，它给出了一条基本惯例，就是将一切都看作是文件。

8.1.1 可加载模块

与其他操作系统不同，Linux操作系统能够在运行时添加或移除内核组件。Linux是单内核（monolithic kernel）结构，内部包含了非常完善的接口，可以在启动后实现设备驱动程序模块的动态加载或者卸载。这种特性不仅为用户增加了更加灵活的操作手段，而且对于设备驱动程序开发人员来说也具有非凡的价值。假设所开发的设备驱动程序模块运行良好，则在开发阶段而不是在出现变化要每次重新启动内核时，随意地从运行时内核动态地完成驱动程序模块的加载和卸载。

对于嵌入式系统来说，可加载模块非常重要，它增强了现场升级的能力，模块本身也可以在系统运行中进行升级而不需要重新启动操作系统。模块可以保存在计算机的某些媒介上而不一定是系统启动设备上，系统的启动设备往往会受到空间的限制。

当然，设备驱动程序也能够静态地构建到操作系统内核之中，且对于很多设备而言，这是非常恰当的做法。举例说明，假设操作系统内核配置要求从网络上通过NFS服务器挂载根文件系统，在这种情况下，要配置与网络相关的设备驱动程序（例如TCP/IP以及网络硬件接口驱动程序等），使其构建到主内核映像中，这样，当操作系统引导并且加载内核之后，网络设备就会被自动初始化以便于实现远程根文件的挂载。当然，也可以利用第6章所描述的初始ramdisk技术，将这些设备驱动程序实现静态构建，让其作为真正内核的一部分。这样，在加载初始ramdisk内核映像的同时，完成必要模块和脚本的加载。

可加载模块会在内核启动后安装，启动脚本用来加载设备驱动程序模块。在必要的情况下，这些设备驱动程序模块也可以实现按需加载。当有服务请求需要一个特殊的模块时，内核需要具有请求模块的能力。

从术语上说，当讨论内核驱动程序模块时，并没有统一的称谓。在讨论可加载内核模块的时候，很多称谓可以互换。在本章及后续章节中，设备驱动程序（device driver）、可加载内核模块（Loadable Kernel Module, LKM）、可加载模块（loadable module）和模块（module）都指同一概念，即可加载内核设备驱动程序模块。

8.1.2 设备驱动程序的体系结构

UNIX/Linux系统的开发人员对基本的Linux设备驱动程序模型应该非常熟悉。虽然设备驱动程序模块还在不断地发展，但是一些基本的结构却并没有随着UNIX/Linux操作系统的演变而发生变化。总的来说，设备驱动程序可以分为两大类：字符设备（character device）和块设备（block device）。字符设备处理顺序数据的串行流，例如串行通信接口或者键盘等；块设备能够从某些具

有地址的媒介上的随机位置读取数据块，或向某些具有地址的媒介上的随机位置写入数据块，例如硬盘驱动器或者软盘驱动器。

8.1.3 最小设备驱动程序示例

由于Linux支持可加载设备驱动程序，所以可以相对容易地描述一个简单设备驱动程序的框架。代码清单8-1给出了一个可加载设备驱动程序的示例，这个模块具有最简化结构框架，能够由运行时内核进行动态的加载或卸载。

代码清单8-1 最小的设备驱动程序框架

```
/* Example Minimal Character Device Driver */
#include <linux/module.h>

static int __init hello_init(void)
{
    printk("Hello Example Init\n");

    return 0;
}

static void __exit hello_exit(void)
{
    printk("Hello Example Exit\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_AUTHOR("Chris Hallinan");
MODULE_DESCRIPTION("Hello World Example");
MODULE_LICENSE("GPL");
```

代码清单8-1给出了设备驱动程序的基本轮廓，它已经具备了可加载内核驱动程序模块所需要的结构，并且包含了执行初始化以及退出的例程。仔细阅读这个源代码，它包含了一些非常重要的高级概念，这些概念对于设备驱动程序的开发将非常有益。

设备驱动程序实际上是一种非常特殊的二进制模块。与可以独立执行的二进制可执行应用程序不同，设备驱动程序模块无法通过命令提示符简单地执行。在2.6版本的内核系列中，这些二进制模块都具有某些特殊的“内核对象”格式。一旦实现了适当的构建工作，内核驱动程序模块就是具有.ko扩展名的二进制对象文件。用于生成.ko文件的构建步骤以及编译器选项比较复杂。本书将概述利用Linux内核构建系统实现内核驱动程序构建的基本步骤，你并不需要成为Linux内核构建系统的专家，这超出了本书的讨论范围。

8.1.4 模块构建的基础设施

设备驱动程序必须在目标操作系统内核上完成相应的编译工作。虽然可以在当前的操作系统内核上加载并执行使用不同内核版本构建的设备驱动程序，但是这么做会带来潜在的风险，除非你能够确认当前加载的设备驱动程序与运行的新内核之间没有任何内在关联。其实最简单的方法，就是在目标操作系统内核源码树下完成设备驱动程序的构建。这样能够确保当开发人员更改内核设置时，其自定义的设备驱动程序可以在正确的内核配置下自动实现重新构建。当然，也可以在内核源码树之外实现设备驱动程序的构建。然而，这么做的话，开发人员必须确保设备驱动程序的构建配置与你将目标操作系统内核保持同步。为了能够实现正确的构建，通常需要正确设备编译器参数开关，设置内核头文件的位置以及内核配置的选项。

对于代码清单8-1给出的设备驱动程序示例，下面这些变化对Linux内核源码树做以下修改，以正确构建。我们详细解释每一步骤的操作。

(1) 首先从Linux源码树的顶层路径开始，在`.../drivers/char`下创建名为`examples`的目录；

(2) 在内核配置文件中增加一项，以便构建内核驱动程序`examples`，并指定该驱动程序是可加载驱动程序还是静态内置驱动程序；

(3) 在`.../drivers/char/Makefile`路径下增加一个新的子目录`examples`，并且根据第(2)步相应的选项来设置；

(4) 在这个新的`examples`路径下创建`makefile`，并且根据第(2)步设置选项增加一个名为`hello.o`的模块对象；

(5) 最后，根据代码清单8-1的代码创建驱动程序源代码`hello.c`。

在`.../drivers/char`目录下创建`examples`目录的道理显而易见。当创建完这个目录后，需要在其中创建两个文件：驱动程序模块的源代码文件和`makefile`。源代码文件根据代码清单8-1来创建，而`makefile`的内容则非常简单，只要包含下面一行代码即可：

```
obj-$(CONFIG_EXAMPLES) += hello1.o
```

接下来的步骤是需要在内核配置文件中增加新的项。这项工作稍微繁琐一些，代码清单8-2给出了部分片段。当从最新Linux操作系统应用`.../drivers/char/Kconfig`文件时，需要增加必要的针对`examples`的配置选项。对不甚了解`diff/patch`格式的人来说，代码清单8-2中含“+”前缀的每一行代码都是注释的说明文字。

代码清单8-2 针对示例的Kconfig文件片段

```
diff -u ~/base/linux-2.6.14/drivers/char/Kconfig
./drivers/char/Kconfig
--- ~/base/linux-2.6.14/drivers/char/Kconfig
+++ ./drivers/char/Kconfig
@@ -4,6 +4,12 @@
 menu "Character devices"
```

```

+config EXAMPLES
+    tristate "Enable Examples"
+    default M
+    ---help---
+    Enable compilation option for driver examples
+
config VT
    bool "Virtual terminal" if EMBEDDED
    select INPUT

```

当设置完Kconfig文件并且保存在.../drivers/char子目录下之后，上述的代码片段最终在当前操作系统内核配置选项中增加了新的一项CONFIG_EXAMPLES。回顾第4章对构建Linux内核的讨论，相应的配置程序需要通过下面的命令才能进行调用（假设当前的示例应用在ARM体系结构下）：

```
$ make ARCH=ARM CROSS_COMPILE=xscale_be- gconfig
```

当使用类似上面的命令调用了内核配置程序之后，Character devices菜单下就会增加Enable Examples这个新的配置选项。由于这个选项可以有3种设置选择，所以内核开发工程师可以从中进行选择：

- ☐ (N)，不编译examples；
- ☐ (Y)，编译examples并且与最终的内核映像进行链接；
- ☐ (M)，模块化设置，将examples编译为动态可加载模块。

图8-1演示了在gconfig工具中增加新配置选项的方法。复选框中的(-)表示选择了(M)选项，如图中最右边的M列所示^①。复选框中的复选标记表示选择了(Y)选项，表明设备驱动程序模块需要编译为严格意义上的内核的一部分。如果不选择任何复选框，则表明不选择相应的选项。

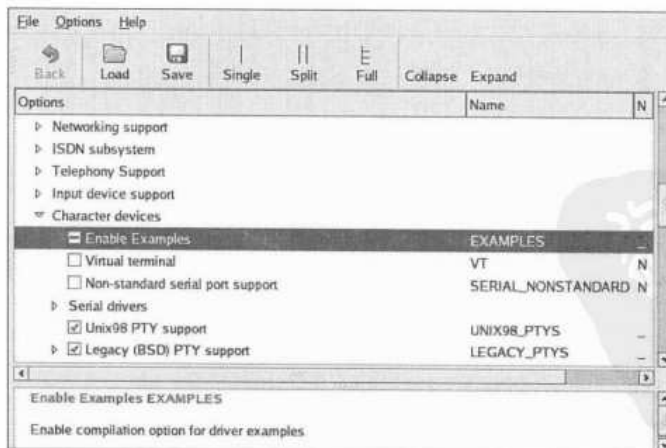


图8-1 在gconfig工具中设置Enable Examples模块选项

^① 图中未见。——译者注

这样，就完成了可以编译examples设备驱动程序模块的配置选项的设置工作。接下来需要修改.../drivers/char路径下的makefile，来指导内核构建系统根据内核配置选项CONFIG_EXAMPLES的设置完成examples的构建工作。代码清单8-3包含了根据当前Linux操作系统发行版修改的makefile的片段。

代码清单8-3 examples的makefile片段

```
diff -u ~/base/linux-2.6.14/drivers/char/Makefile
./drivers/char/Makefile
--- ~/base/linux-2.6.14/drivers/char/Makefile
+++ ./drivers/char/Makefile
@@ -88,6 +88,7 @@
obj-$(CONFIG_DRM) += drm/
obj-$(CONFIG_PCMCIA) += pcmcia/
obj-$(CONFIG_IPMI_HANDLER) += ipmi/
+obj-$(CONFIG_EXAMPLES) += examples/

obj-$(CONFIG_HANGCHECK_TIMER) += hangcheck-timer.o
```

在代码清单8-3给出的makefile片段中，具有“+”前缀的那一行代码是新增加的，这个makefile需要保存在.../drivers/char路径下，其他代码则指明了相应的工具决定这一行新代码插到哪里。新创建的examples目录加到了目录列表的最后，且makefile可以搜索到它。这个目录看起来是一个非常适合的地方。除了为保持一致性和可读性，目录的位置是无关紧要的。

当完成这些步骤后，构建examples设备驱动程序的基础设施就准备好了。利用这个步骤完成构建工作的好处是，不管什么时候调用内核构建工作，相应的驱动程序都会被自动地构建。只要根据代码清单8-3所定义的配置选项进行选择（M或者Y），就可以完成设备驱动程序的构建。

构建任意的ARM系统，相应的命令行为：

```
$ make ARCH=arm CROSS_COMPILE=xscale_be- modules
```

代码清单8-4给出了在模块上编辑会话后的构建输出信息（所有其他模块都已经构建在了内核源码树中）。

代码清单8-4 模块构建的输出

```
$ make ARCH=arm CROSS_COMPILE=xscale_be- modules
CHK      include/linux/version.h
make[1]: 'arch/arm/kernel/asm-offsets.s' is up to date.
make[1]: 'include/asm-arm/mach-types.h' is up to date.
CC [M]   drivers/char/examples/hello1.o
Building modules, stage 2.
MODPOST
LD [M]   drivers/char/examples/hello1.ko
```

8.1.5 安装设备驱动程序

现在，设备驱动程序已经构建好，接下来要将其加载到运行的内核中，再从内核中卸载它，从而考察它的行为。在加载设备驱动程序模块之前，需要将相应的文件复制到目标系统中合适的

位置。虽然理论上可以把设备驱动程序模块复制到系统中任意位置上,但是习惯的做法是将其放在便于内核模块进行构建、且易于迁移到运行中的Linux系统的合适位置。这样做,可以非常容易地使内核构建系统完成设备驱动程序模块的编写工作。`makefile`定义的编译目标`modules_install`将自动地完成设备驱动程序模块的安装工作。作为开发人员,只要简单地进行默认目标路径的设置就可以了。

在进行标准的Linux操作系统工作站版本安装时,细心的你可能已经发现所有设备驱动程序模块都保存在`/lib/modules/<kernel-version>/...`路径下,且按照类似于Linux操作系统内核源码树中设备驱动程序目录层次结构的次序分别保存^①。其中的`<kernel_version>`可以通过在目标系统上执行`uname -r`命令得到。默认情况下,如果不提供内核构建系统的安装位置,则设备驱动程序模块会安装到Linux操作系统工作站的`/lib/modules/...`目录下。这可能不是大家想要的结果。其实,完全可以在`home`目录下给出一个临时的目录,然后手工地将所有模块复制到目标系统的相应目录下。还有一种方法,如果嵌入式目标系统使用NFS挂载到本地开发工作站的一个目录上,则可以直接将驱动程序模块安装到指定的目标文件系统内。接下来的例子将演示这一过程。

```
$ make ARCH=arm CROSS_COMPILE=xscale_be- \
  INSTALL_MOD_PATH=/home/chris/sandbox/coyote-target \
  modules_install
```

上面的命令将所有模块都放置在了`coyote-target`目录下,这是一个利用NFS挂载到目标系统根文件下的示例系统^②。

8.1.6 加载设备驱动程序模块

至此,所有必需的步骤都已经完成,现在要加载并测试前面开发的设备驱动程序模块。代码清单8-5演示了在嵌入式系统上加载设备驱动程序模块以及卸载相应模块的输出信息。

代码清单8-5 加载/卸载设备驱动程序模块

```
$ modprobe hello1          <<< Load the driver
Hello Example Init
$ modprobe -r hello1       <<< Unload the driver
Hello Example Exit
$
```

你应该能够由上述输出信息联系到代码清单8-1所示的设备驱动程序模块的源代码。这个模块除了通过`printk()`向内核日志系统输出信息外,没有做其他任何工作,该信息就是在控制台看到的输出信息^③。当设备驱动程序模块被加载时,就会调用模块的初始化函数。初始化函数使

① 这个路径是Red Hat Linux或者Fedora Core操作系统使用的路径,同样也是在本章后面介绍的文件系统层次结构标准(File System Hierarchy Standard)所要求的路径。其他Linux操作系统发行版可能存在差别。

② 第12章会详细介绍如何挂载NFS系统以及目标系统的方法。

③ 如果你没有看到控制台中相应的输出信息,则有可能屏蔽了系统日志记载或者降低了控制台日志信息记载的级别。在第14章中将详细描述。

用`module_init()`宏在模块加载时执行, `module_init()`的声明如下:

```
module_init(hello_init);
```

在初始化函数中, 我们仅简单输出了应有的“hello”消息然后就返回了。在实际的设备驱动程序模块中, 要完成初始资源的分配以及硬件设备的初始化工作。同样地, 当卸载驱动程序模块(使用`modprobe -r`命令)时, 也会调用模块的退出函数。在代码清单8-1中可以发现, 退出函数使用`module_exit()`宏指定。

至此, 这个设备驱动框架程序已经能够加载到实际的操作系统内核中使用了。后面将介绍可加载设备驱动程序模块的其他功能, 从而演示用户空间的程序如何与设备驱动程序模块进行交互。

8.2 模块实用程序

代码清单8-5中已经简单地引入了驱动程序模块的实用程序, 其中使用了`modprobe`实用程序实现了从Linux内核加载和卸载驱动程序模块。Linux操作系统为用户提供了一系列小实用程序, 用于管理设备驱动程序模块。本节将介绍这些实用程序的基本用法。你可以使用`man`命令来了解这些实用程序, 以获取更详尽的信息。实际上, 对Linux可加载设备驱动程序模块具有浓厚兴趣的读者应该仔细查看这些实用程序的源代码。本章最后的“参考资源”提供了一些有用的信息。

8

8.2.1 insmod

`insmod`实用程序是将模块加载到运行中的内核的最简单方式, 只要给出完整的路径和文件名, `insmod`就能够完成工作, 例如:

```
$ insmod /lib/modules/2.6.14/kernel/drivers/char/examples/hello1.ko
```

上面的命令将`hello1.ko`模块加载到内核中。相应的输出信息与代码清单8-5的相同, 就是Hello消息。`insmod`实用程序很简单, 因为它不要求或接受任何选项, 所需要的只是模块文件所在的完整路径信息, 因为它不具备搜索相对路径的能力。大多数情况下, 应该使用`modprobe`实用程序, 它具有丰富的功能和特性。

8.2.2 模块参数

很多设备驱动程序模块在加载过程中可以接受参数来定义其行为。例如启用调试模式、设置verbose报告或指定特定于模块的选项。`insmod`实用程序可以通过在给定的模块名称之后增加必要的参数(某些环境下也成为选项)。代码清单8-6给出了已修改的`hello1.c`示例文件, 它增加了一个模块参数允许按照调试模式加载模块。

代码清单8-6 设备驱动程序示例——参数设置

```
/* Example Minimal Character Device Driver */
#include <linux/module.h>

static int debug_enable = 0;      /* Added driver parameter */
```

```

module_param(debug_enable, int, 0); /* and these 2 lines */
MODULE_PARM_DESC(debug_enable, "Enable module debug mode.");
static int __init hello_init(void)
{
    /* Now print value of new module parameter */
    printk("Hello Example Init - debug mode is %s\n",
           debug_enable ? "enabled" : "disabled")

    return 0;
}

static void __exit hello_exit(void)
{
    printk("Hello Example Exit\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_AUTHOR("Chris Hallinan");
MODULE_DESCRIPTION("Hello World Example");
MODULE_LICENSE("GPL");

```

与前面相比，代码清单8-6的代码增加了三行新的代码。第一行代码声明了一个静态的整数变量作为调试模式的标志，第二行代码是引用了.../include/linux/moduleparam.h文件中定义的一个宏，把相应的模块参数注册到内核模块子系统中。第三行代码同样也是一个宏调用，向内核模块子系统注册了一个字符串，这个字符串与前面注册的参数相关。等后面介绍modinfo命令时，大家就会明白在这里注册字符串的意义了。

假设，现在还使用insmod实用程序加载我们的示例模块，并在命令行中增加debug_enable选项，那么应该能够看到相应的输出信息，代码清单8-6给出加载hello1.c定义的内核驱动程序的执行过程：

```

$ insmod /lib/modules/.../examples/hello1.ko debug_enable=1
Hello Example Init - debug mode is enabled

```

如果忽略可选的内核参数，则过程如下：

```

$ insmod /lib/modules/.../examples/hello1.ko
Hello Example Init - debug mode is disabled

```

8.2.3 lsmod

lsmod实用程序其实也很少使用，它的功能就是简单地列出已经加载到内核中所有的内核驱动程序模块。当前版本的Linux操作系统中，不需要为lsmod实用程序提供任何参数，这个小实用程序就能够带格式地输出/proc/modules下的信息^①。代码清单8-7给出了使用lsmod的基本过程以及相应的结果。

^① /proc/modules是proc文件系统的一部分，详细信息见第9章。

代码清单8-7 lsmod输出信息的格式

```
$ lsmod
Module                Size  Used by
ext3                  121096  0
jbd                   49656   1 ext3
loop                  12712   0
hello1                1412    0
$
```

注意上面最右边Used by列的内容。这一列信息表示当前的设备驱动程序模块由谁使用，并给出了相应的依赖关系。例如，上面的jbd模块（日志文件系统模块）正在被ext3模块所使用，默认情况下，日志文件系统在很多流行的Linux桌面发行版中都能找到。这里的依赖关系意味着ext3设备驱动程序依赖于jbd模块。

8.2.4 modprobe

使用modprobe是管理可加载设备驱动程序模块的上佳选择。从代码清单8-7中可以发现ext3与jbd之间的关系，就是ext3模块依赖于jbd模块。利用modprobe实用程序能够发现这个关系，并按照正确的次序加载这样有依赖关系的设备驱动程序模块。下面的命令将完成jbd.ko和ext3.ko设备驱动程序模块的加载：

```
$ modprobe ext3
```

modprobe实用程序有若干个命令行选项，可以控制其行为。如前所述，利用modprobe实用程序可以卸载设备驱动程序模块，同时也会卸载一个给定模块所依赖的模块。例如下面的命令，不仅会卸载jbd.ko模块，还会卸载ext3.ko：

```
$ modprobe -r ext3
```

modprobe实用程序由名称为modprobe.conf的配置文件驱动。利用这个配置文件，系统开发人员可以将若干硬件设备与设备驱动程序相关联。对于简单的嵌入式系统，modprobe.conf的内容可以为空或者只有简单的几行代码。当modprobe.conf不存在或无效时，modprobe实用程序会按照一组默认的规则编译。调用modprobe时仅使用-c选项，将显示modprobe实用程序所使用的这组默认规则。

代码清单8-8给出了modprobe.conf，在具有两个以太网接口的Linux操作系统中也可能发现类似内容的配置文件。这两个以太网接口，一个是基于Prism2芯片组的无线适配器，另一个是一般的PCI以太网卡。这个系统也包含基于集成了Intel音频芯片组的声卡子系统。

代码清单8-8 典型的modprobe.conf 文件

```
$ cat /etc/modprobe.conf
alias eth1 orinoci_pci
options eth1 orinoco_debug=9
alias eth0 e100
alias snd-card-0 snd-intel8x0
options snd-card-0 index=0
$
```

当内核启动并发现无线网络芯片组时，该配置文件将指导modprobe加载orinoco_pci设备驱动程序模块，将其与内核设备eth1绑定，同时将可选的模块参数orinoco_debug=9传递到设备驱动程序中。同样的工作也会在发现声卡时进行。要注意与声卡设备驱动程序snd-intel8x0相关的参数。

8.2.5 depmod

你可能会问，modprobe是如何知道设备驱动程序模块之间彼此的依赖性呢？depmod实用程序在这个过程中起了关键的作用。当执行modprobe实用程序时，它首先搜索名为modules.dep的文件，这个文件与相应的模块保存在同一个位置。depmod实用程序就是用来创建这个模块依赖性文件的。

这个文件中保存了所有内核构建系统需要配置的模块的列表，同时保存了模块之间的依赖信息。它的文件格式非常简单，每个设备驱动程序模块占据文件的一行。如果某个模块有依赖成员，这些依赖成员会按次序列在模块名之后。如代码清单8-7所示，ext3模块依赖于jbd模块，所以在modules.dep文件中应该有这样一行：

```
ext3.ko: jbd.ko
```

在实际应用中，每个设备驱动程序模块名的前面都有其文件系统的绝对路径，这么做主要是为了避免出现二义性差错。上面为了便于大家阅读忽略了路径信息。对于较复杂的依赖关系（例如声卡驱动程序）会类似下面这样：

```
snd-intel8x0.ko: snd-ac97-codec.ko snd-pcm.ko snd-timer.ko \
snd.ko soundcore.ko snd-page-alloc.ko
```

这里为了便于阅读，再次去掉了各个组件的路径信息。在modules.dep文件中出现的每个模块都使用绝对文件名，包含了完整的路径信息，并且每个都占据一行。上面的例子把它们排在两行，完全是受书的版面限制。

一般来说，depmod都是在内核构建过程中自动运行的。不过，在交叉开发环境中，你必须使用交叉版的depmod实用程序，这样就知道如何读取以目标体系结构的本地格式编译的模块。大多数嵌入式系统的发行版都使用这种方法：在每次启动系统时通过init脚本中运行depmod，以保证得到最新的模块依赖关系。

8.2.6 rmmod

这个实用程序一般也较少使用。它的功能很简单，就是将模块从当前运行的内核中卸载。rmmod传递的参数就是要卸载的模块名称，不需要文件的绝对路径名称或者文件扩展名，例如：

```
$ rmmod hello1
Hello Example Exit
```

使用rmmod时唯一需要着重理解的是，它在卸载驱动程序模块时，将执行模块的*_exit()函数，如代码清单8-1和代码清单8-6给出的示例。

也应注意，与modprobe不同，rmmod不卸载依赖模块，所以最好使用modprobe -r命令来卸载模块。

8.2.7 modinfo

你可能还记得代码清单8-1中的最后三行，同样内容也出现在代码清单8-6中。这些宏用于在二进制模块放置一些标志，以便于模块的管理和维护。代码清单8-9是使用modinfo实用程序获取hello1.ko模块的输出信息。

代码清单8-9 modinfo的输出

```
$ modinfo hello1
filename:      /lib/modules/.../char/examples/hello1.ko
author:        Chris Hallinan
description:    Hello World Example
license:        GPL
vermagic:       2.6.14 ARMv5 gcc-3.3
depends:
parm:          debug_enable:Enable module debug mode. (int)
$
```

一开始的内容很简单，这是这个设备驱动程序模块的完整文件名称，包含了路径名称。不过为了阅读便于，这里再次缩略了文件的路径名。接下来几行是代码清单8-6中定义的宏给出的直接结构，包含文件名称、作者、授权信息等。这些是模块实用程序所使用的简单的标志，它们并不影响内核驱动程序本身的功能。通过man命令以及modinfo实用程序的源代码可以获取更多有关该实用程序的信息。

modinfo实用程序的一个非常有用的功能是，可以了解内核驱动程序模块支持哪些参数。在代码清单8-9中，我们看到该模块仅支持一个参数，这个参数正是在代码清单8-6的代码中增加的参数debug_enable。这个代码清单中给出了参数的名称、类型（本例中为int）以及描述性文本字段，描述性文本字段使用MODULE_PARM_DESC()宏进行定义。这种方法很方便，特别适用于不易获得源代码的模块。

8.3 驱动程序方法

到目前为止，我们已经介绍了很多模块实用程序的背景知识。在后面几节中，还会介绍一些设备驱动程序与用户空间程序（也就是应用程序代码）之间进行交互的基本机制。

前面两节介绍了两个基本的函数：一个用于模块的初始化，另一个用于处理模块的退出过程。回顾代码清单8-1，其中的module_init()和module_exit()就是完成模块初始化与退出的函数。我们已经了解这两个函数在模块加载到运行中的内核或从内核中卸载时被调用，现在需要其他一些函数来充当设备驱动程序和编写的应用程序之间的接口。不过，一定要牢记，使用设备驱动程序有两个最重要的原因：一是避免用户在内核空间编写代码，因为这样非常危险；二是为与硬件或内核级设备通信提供统一的方法。

8.3.1 驱动程序文件系统操作

设备驱动程序模块加载到活动的内核后，我们必须采取的第一项活动是为驱动程序准备相关的操作。`open()`方法就用于这个目的。当设备驱动程序被打开后，需要相应的函数来完成数据的读写操作。`release()`函数可以用来清理完成相应操作之后的现场（通常就是关闭设备接口）。最后，还需要一个特定的系统调用，来完成与设备驱动程序之间的非标准通信，这个函数一般定义为`ioctl()`。代码清单8-10中为前面介绍的示例中增加了这些基本内容。

代码清单8-10 增加文件系统操作到hello.c

```
#include <linux/module.h>
#include <linux/fs.h>

#define HELLO_MAJOR 234

static int debug_enable = 0;
module_param(debug_enable, int, 0);
MODULE_PARM_DESC(debug_enable, "Enable module debug mode.");

struct file_operations hello_fops;

static int hello_open(struct inode *inode, struct file *file)
{
    printk("hello_open: successful\n");
    return 0;
}

static int hello_release(struct inode *inode, struct file *file)
{
    printk("hello_release: successful\n");
    return 0;
}

static ssize_t hello_read(struct file *file, char *buf, size_t count,
                          loff_t *ptr)
{
    printk("hello_read: returning zero bytes\n");
    return 0;
}

static ssize_t hello_write(struct file *file, const char *buf,
```

```

        size_t count, loff_t * ppos)
{
    printk("hello_read: accepting zero bytes\n");
    return 0;
}

static int hello_ioctl(struct inode *inode, struct file *file,
                      unsigned int cmd, unsigned long arg)
{
    printk("hello_ioctl: cmd=%ld, arg=%ld\n", cmd, arg);
    return 0;
}

static int __init hello_init(void)
{
    int ret;
    printk("Hello Example Init - debug mode is %s\n",
           debug_enable ? "enabled" : "disabled");
    ret = register_chrdev(HELLO_MAJOR, "hello1", &hello_fops);
    if (ret < 0) {
        printk("Error registering hello device\n");
        goto hello_fail1;
    }
    printk("Hello: registered module successfully!\n");

    /* Init processing here... */

    return 0;

hello_fail1:
    return ret;
}

static void __exit hello_exit(void)
{
    printk("Hello Example Exit\n");
}

struct file_operations hello_fops = {
    owner:    THIS_MODULE,
    read:     hello_read,
    write:    hello_write,
    ioctl:    hello_ioctl,
    open:     hello_open,
    release:  hello_release,
}
```



```
};

module_init(hello_init);
module_exit(hello_exit);

MODULE_AUTHOR("Chris Hallinan");
MODULE_DESCRIPTION("Hello World Example");
MODULE_LICENSE("GPL");
```

这个扩展后的设备驱动程序示例包含了很多新的代码行。在最开始的地方，增加了一个新的内核头文件，用于获取文件系统操作的定义。同样，还定义了一个设备驱动程序的主设备号。（设备驱动程序的设计者注意，这不是定义设备驱动程序版本号的恰当方式，详见Linux内核文档.../Documentation/devices.txt，或者指导定义设备驱动程序版本号的优秀图书。对于这个简单的示例，我们简单地选择了一种在我们的系统中不会使用的方法。）

接下来可以看到，代码中定义了4个新的函数，分别为完成模块的打开（open）、关闭（close）、读取（read）和写入（write）。基于良好的编程习惯，这里定义这些函数时采用了一致的命名原则，主要是为了避免与内核中的其他子系统的设备驱动程序相冲突。这里定义的函数名称分别为hello_open()、hello_release()、hello_read()和hello_write()。为了简便起见，这些函数仅向内核日志子系统输出一些信息，并没有针对任何硬件设备的实质操作。

注意，我们在hello_init()函数中还增加了一个新的函数调用，这行代码把设备驱动程序注册到内核。通过这个函数调用，我们传递了一个包含所需要的方法的结构指针。内核使用这个数据结构，也就是类型为struct file_operations的结构，将特定的设备函数与来自相应文件系统的请求绑定在一起。当用户应用程序打开某个由设备驱动程序表示的设备，并进行读取数据（使用read()函数）的操作时，相应的文件系统就会将read()函数关联到模块的hello_read()函数。后面内容将详细探讨这个过程。

8.3.2 设备节点与mknod

为了能够理解应用程序是如何实现将其请求与设备驱动程序所表示的设备绑定在一起的，首先需要理解设备节点的概念。设备节点是使Linux操作系统为了表示某个设备而定义的一种特殊文件类型。几乎所有的Linux操作系统发行版都保留了设备节点这个概念，并分配到名为/dev目录的统一位置（文件系统层次结构标准^①）。Linux定义了一个实用程序mknod，用于创建设备节点。

通过学习创建设备节点的示例有助于更好地阐述设备节点的功能及其包含的信息。这里还是利用前面简单的设备驱动程序示例，通过下面的命令练习创建设备节点：

```
$ mknod /dev/hello1 c 234 0
```

^① 请参考本章“参考资源”。

在目标嵌入式系统上运行这条命令之后，在系统中就会增加一个新的文件/dev/hello1，用来表示设备驱动程序的设备节点。如果在控制台中键入下面的命令：

```
$ ls -l /dev/hello1
crw-r--r-- 1 root root 234, 0 Jul 14 2005 /dev/hello1
```

就可以查看设备驱动程序节点的相关信息。

使用mknod时，需要将文件名称、类型、主版本号、辅助版本号等信息传递给该命令。当然，在本例中选择的设备文件名称是hello1，由于是以字符变量为基础的设备驱动程序，因此我们使用c表示设备节点的类型。主版本号是234，辅助版本号是0，这些都很容易理解。

就设备节点本身而言，它只不过是文件系统中的文件。然而，由于设备节点具有很特殊的作用，因而被用于绑定安装的设备驱动程序模块。如果应用程序使用open()函数，并将设备节点作为其路径参数，内核会搜索所有使用与该设备节点匹配的主版本号（这里是234）进行注册的有效设备驱动程序。这种机制就是内核将特定设备与相应设备节点相互关联的方式。

大多数C程序员都知道，open()函数的系统调用（或其变体）将返回一个引用（文件描述符），应用程序通过该引用来识别相应的文件，如完成文件的读写、关闭等操作。在进行各种操作的同时，对应的设备文件系统也要完成相应的操作。

辅助版本号主要就是用来识别同一个设备驱动程序的不同版本或者子设备驱动程序。一般来说，操作系统并不使用这个信息，它仅仅传递给对应的设备驱动程序，而设备驱动程序根据辅助版本号判断是否匹配。例如，假设当前系统具有一块多路串口卡，主版本号定义了设备驱动程序，而辅助版本号则可能对应多路串口卡中由同一设备驱动程序处理的不同的串行通信端口。如果对此方面有兴趣，可以选择详细介绍设备驱动程序的图书^①进行深入的了解。

8

8.4 汇总

至此，我们已经完成了一个设备驱动程序的基本框架，并且能够成功加载并练习这个程序了。代码清单8-11给出了一个简单的用户空间应用程序的示例，以此来运用设备驱动程序。前面已经介绍了如何加载设备驱动程序模块，就是简单地进行编译，并使用make modules_install命令将驱动程序模块引入文件系统。

代码清单8-11 运用设备驱动程序

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
```

① 推荐Sreekrishnan Venkateswaran著《精通Linux设备驱动程序》（人民邮电出版社即将出版）。——编者注

```

/* Our file descriptor */
int fd;
int rc = 0;
char *rd_buf[16];

printf("%s: entered\n", argv[0]);

/* Open the device */
fd = open("/dev/hello1", O_RDWR);
if ( fd == -1 ) {
    perror("open failed");
    rc = fd;
    exit(-1);
}
printf("%s: open: successful\n", argv[0]);
/* Issue a read */
rc = read(fd, rd_buf, 0);
if ( rc == -1 ) {
    perror("read failed");
    close(fd);
    exit(-1);
}
printf("%s: read: returning %d bytes!\n", argv[0], rc);

close(fd);
return 0;
}

```

这是一段非常简单的源代码，需要在ARM XScale系统上进行编译，文件中演示了如何通过设备节点将用户应用程序与开发的设备驱动程序绑定在一起。与前面开发的设备驱动程序类似，它没有进行实质性的工作，但是已经使用了代码清单8-10的那几个函数，演示了完整的概念。

首先，在之前上调用创建的设备节点`open()`函数^①，如果设备能够成功地打开（`open`），则在控制台窗口中输出相应的文本信息。接下来，通过`read()`函数读取某些信息，如果成功的话，则又将在控制台窗口输出相应的文本信息。注意，就内核而言，如果仅仅读取了0字节信息，这也是可以接受的一种结果。在实际的工作中，还需要关注是否已经到文件末尾或者数据是否溢出等意外情况。当完成所有工作之后，利用`close()`函数关闭设备，并且退出。代码清单8-12给出在ARM XScale处理器平台上运行该例子的结果。

代码清单8-12 运行设备驱动程序示例

```

$ modprobe hello1
Hello Example Init - debug mode is disabled
Hello: registered module successfully!
$ ./use-hello
./use-hello: entered
./use-hello: open: successful

```

① 实际上，`open()`函数是一个C语言的包装函数（wrapper function），实际调用的函数是`sys_open()`。

```
./use-hello: read: returning zero bytes!
$
```

8.5 设备驱动程序与 GPL

目前，关于Linux设备驱动程序以及如何让设备驱动程序符合GPL还存在非常多的讨论甚至争论。首先，大多数人都了解下面的基本原则：如果开发的设备驱动程序（或者各种软件）是基于现有的GPL软件，哪怕是部分基于现有的GPL软件，那么这种软件都称作衍生作品（*derived work*）。例如，以一个现有的Linux设备驱动程序为起点，对其进行适度的修改以适应新项目的需要，那么，开发这个设备驱动程序的工作就是衍生作品。作为开发人员，必须遵守GPL的规定，将这个修改的设备驱动程序按照GPL的条款进行发布。

正是这一点引起了大家的争论。首先是免责声明。这并不是一种合法的做法，并且作者也不可能是法律工作者。这里面涉及的一些概念并没有在法庭上得到验证。目前，合法和开源社区的主流观点是，如果能够证明开发工作是独立完成的^①，并且开发出来的设备驱动程序与Linux内核之间的关系并不十分紧密，那么开发人员可以以他们认为合适的方式自由发布其开发成果。不过，如果开发人员对内核进行了修改，以满足内核驱动程序的某种特殊需要，那么这种工作就视为衍生作品，需要遵守GPL。

在开源社区内存在大量涉及此类内容的信息，并且信息的数量还在持续增长。可能在不久的将来，这些概念会在法律层面得到验证，并且确立先例。每个人都在猜测，这需要多长时间。如果你对围绕Linux操作系统以及开源运动相关的法律问题感兴趣，并想对此有更好的理解，可以浏览www.open-bar.org网站。

8.6 小结

本章从较高层面介绍了Linux设备驱动程序的基础知识，并且介绍了如何将设备驱动程序正确引入Linux操作系统。有了这些基础知识，刚刚理解设备驱动程序的读者就可以通过学习一些深层次的内容进一步了解Linux设备驱动程序系统的开发与应用。查看“参考资源”中的文档或者图书。

- 设备驱动程序合理地分离了没有权限的用户应用程序与关键的内核资源（硬件和其他设备），并且提供众所周知的统一应用程序接口；
- 用来加载设备驱动程序模块的最小基础设施只需要短短的几行代码，本章对此进行了介绍，并帮助你迅速建立设备驱动程序模块的基本概念；
- 设备驱动程序模块可以配置为可加载模块，在Linux内核启动后的加载运行阶段可以将其插入到内核或从内核卸载；
- 设备驱动程序的实用程序用来管理驱动程序模块的插入、移除以及列表打印等操作。本章详细介绍了这些实用程序的功能；

^① 这种做法并不是开源运动特有的，版权和专利侵权涉及每个开发人员。

- 文件系统的设备节点将用户空间应用程序与设备驱动程序联系起来;
- 设备驱动程序函数实现了在UNIX/Linux设备驱动程序中比较常见的、大家所熟悉的打开、读取、写入以及关闭等功能,本章通过示例介绍了这些函数,包括运用设备驱动程序的简单用户应用程序;
- 本章还介绍了内核设备驱动程序与开源GPL之间的关系。

参考资源

Linux Device Drivers, 3rd Edition

Alessandro Rubini and Jonathan Corbet

O'Reilly Publishing, 2005

《文件系统层次结构标准》

Rusty Russel、Daniel Quinlan和 Christopher Yeoh 编辑

文件系统层次结构标准小组

www.pathname.com/fhs/

Rusty的Linux内核主页

2.6版内核使用的模块实用程序

Rusty Russell

<http://kernel.org/pub/linux/kernel/people/rusty/>



本章内容

- Linux文件系统的概念
- ext2文件系统
- ext3文件系统
- ReiserFS文件系统
- JFFS2文件系统
- cramfs文件系统
- NFS文件系统
- 伪文件系统
- 其他文件系统
- 构建简单的文件系统
- 小结

对于嵌入式开发者来说，部署什么样的文件系统可能是最重要的抉择之一。有些文件系统有助于优化系统性能，而有些文件系统可以优化系统的大小，还有一些文件系统则有助于在设备和电源失效时恢复系统数据。本章将介绍在Linux系统中使用的主要文件系统，并且研究每个文件系统应用到嵌入式系统时的一些特点。这些内容的主要目的并不是详细研究每个文件系统的内部技术细节，而是要研究与这些文件系统开发相关的开发问题和使用特点。感兴趣的读者可以参考本章最后的“参考资源”。

为了进行更深入的探讨，我们从早期桌面Linux版本中使用的最流行的文件系统开始，以介绍ext2文件系统的概念作为基础。接下来，看一下ext2的升级版本ext3文件系统，该文件系统也是如今许多主流桌面Linux系统的默认文件系统。

在介绍一些基本概念后，我们会讨论各个具体的文件系统，包括有助于数据恢复、节省系统存储空间和用在闪存的文件系统的特性。这里也会介绍网络文件系统（NFS），随后还会讨论更重要的文件系统——伪文件系统，包括proc文件系统和sysfs文件系统。

9.1 Linux 文件系统的概念

在介绍每一个文件系统的细节之前，我们先来看一下Linux系统中的数据是如何存储的。第8章讨论了字符设备的结构，一般情况下，字符设备以串行方式存储和获取数据，串口或磁带机是最典型的字符设备。相比之下，块设备会一次存取和获取大小相同的一组数据。例如，一个典型IDE硬盘控制器可以向物理存储介质上指定可寻址地址，一次传送或接收512字节的数据。文件系统是建立在块设备之上的。

分区

在讨论文件系统之前，我们先来介绍系统分区。文件系统是存在于一个物理设备的逻辑分区之上的。数据存储分区中物理设备的最顶端。分区就是对一个物理介质（磁盘、闪存）的逻辑划分，该物理介质上的数据在给定分区类型上按照特定的规则进行组织。物理设备可以只有一个独立分区包含所有可用空间，也可以被划分为多个分区以满足特定需要。分区也可以看作是可以写入一个完整文件系统的逻辑磁盘。

分区和文件系统之间的关系如图9-1所示。

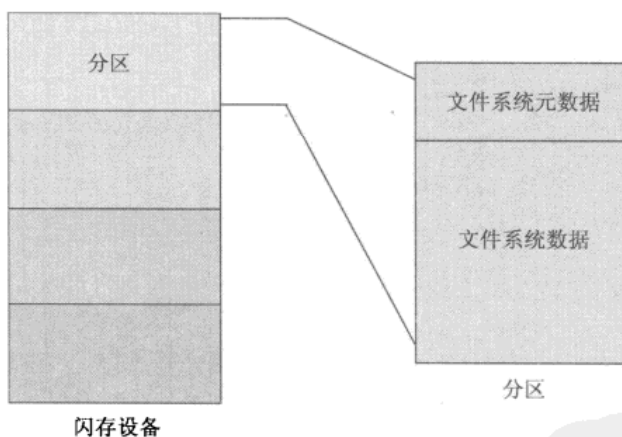


图9-1 分区和文件系统

Linux采用fdisk实用程序进行块设备的分区操作。在许多Linux发行版中可以找到的最新版fdisk已经可以支持90多种分区类型，实际上，只有很少的几种常用于Linux系统。一些常见的分区类型有Linux、FAT32和Linux交换分区。

以一个连接到USB端口的CompactFlash设备为目标，代码清单9-1显示了采用fdisk实用程序的执行结果。在这个特定的目标系统上，USB子系统为CompactFlash物理设备分配的设备节点是/dev/sdb。

代码清单9-1 使用fdisk实用程序显示系统分区信息

```
# fdisk /dev/sdb
```

```
Command (m for help): p
Disk /dev/sdb: 49 MB, 49349120 bytes
4 heads, 32 sectors/track, 753 cylinders
Units = cylinders of 128 * 512 = 65536 bytes
```

Device	Boot	Start	End	Blocks	Id	System
/dev/sdb1	*	1	180	11504	83	Linux
/dev/sdb2		181	360	11520	83	Linux
/dev/sdb3		361	540	11520	83	Linux
/dev/sdb4		541	753	13632	83	Linux

为了便于讨论, 我们使用fdisk实用程序在这个设备上创建了四个分区, 其中一个是有标记的引导分区, 在Boot列中用*号标记, 这是在设备分区表上设置标志的方法, 读者可以从上面的代码清单中看到fdisk所使用的存储设备上的逻辑单元是磁柱(cylinder)^①。在这个设备上, 一个磁柱是64KB。另一方面, Linux声称最小的存储单元是一个逻辑块。从上面的代码清单可以看出, 一个块为1 024字节(1KB)。

当采用上面方法将CompactFlash分区以后, 每个设备分区都可以用你所选择的文件系统类型格式化。当采用给定的文件系统类型格式化一个分区之后, Linux就可以把相应的文件系统挂载到该分区。

9.2 ext2 文件系统

以代码清单9-1构建的分区为例, 我们需要对使用fdisk实用程序创建的分区进行格式化, 为此, 可以使用Linux下的mke2fs实用程序。mke2fs类似于DOS下的格式化命令, 该实用程序用来在特定分区创建一个ext2类型的文件系统。创建其他类型的文件系统也有相应的实用程序。代码清单9-2显示了这一格式化过程的输出信息。

代码清单9-2 使用mke2fs格式化一个分区

```
# mke2fs /dev/sdb1 -L CFlash_Boot_Vol
mke2fs 1.37 (21-Mar-2005)
Filesystem label=CFlash_Boot_Vol
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
2880 inodes, 11504 blocks
575 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=11796480
2 block groups
8192 blocks per group, 8192 fragments per group
1440 inodes per group
Superblock backups stored on blocks:
    8193
```

① 磁柱这个术语来源于磁盘介质的存储单元, 它是由磁盘设备一个指定扇区上一组磁头中的数据组成的, 在这里使用是为了与已存在的文件系统工具相兼容。

```

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 39 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
#

```

代码清单9-2包含了大量关于ext2文件系统的细节,同时提供了一种理解该文件系统使用特性的好方法。注意,这个分区被格式化为ext2类型,其卷标为CFlash_Boot_Vol,该分区被创建为Linux分区(OS type),其块大小为1 024字节,整个空间分为2 880个节点,占用了11 504个块。节点是一个文件的基本数据结构单元。关于ext2文件系统内部结构的详细内容,可以参考本章最后的“参考资源”。

看过代码清单9-2中mke2fs的执行结果之后,我们就可以得知存储设备是如何组织的,也知道了块的大小为1 024字节。如果对于用户的特定应用有必要,mke2fs可以在格式化为一个ext2文件系统时采用不同的块大小。目前所支持的块大小为1 024字节、2 048字节和4 096字节。

块大小总是要服从于系统最佳性能的要求。一方面,由于每个文件必须存放在完整数量的块之中,所以在一个有许多文件的磁盘上,大容量的块会浪费许多空间。在block_size大小为n的块上,其任何剩余碎片都会占用一个完整的块,即使只有一个字节。另一方面,如果采用容量非常小的块,在进行块到文件的映射时,就会增加文件系统对元数据(metadata)的管理成本。选择一个最合适块大小的唯一方法是,在用户特定硬件实现中进行基准测试。

9.2.1 挂载文件系统

创建文件系统之后,就可以在运行的Linux系统中挂载该文件系统了,假定我们已经可以对硬件设备进行访问,并且支持用户特定文件系统类型的内核也已经编译通过,而编译的方法不管是采用在编译过程中添加内核模块或是动态加载内核模块。下面的命令可以把先前创建的ext2文件系统挂载到指定的挂载点:

```
# mount /dev/sdb1 /mnt/flash
```

在这个例子中,假定我们已经在目标Linux机器上创建了名为/mnt/flash的目录。之所以把该目录称为挂载点,是因为这是文件系统层次结构中,安装(挂载)文件系统的地方。我们要把早些时候被内核指定为/dev/sdb1的闪存设备挂载到此处。如果是一个典型的Linux桌面机器(或开发平台),这里需要根用户(root)的权限来执行该命令^①。挂载点的选择可以是文件系统中用户指定的任何地方,挂载后该位置也就成为新挂载设备的根(root)目录。在前面的例子中,如果要引用该闪存设备上的任何文件,都必须在路径前加入前缀/mnt/flash。

mount命令功能强大,具有许多操作选项。mount命令所支持的很多选项都取决于所挂载文件系统的类型。在绝大多数情况下,一个正确格式化的且为内核所知的文件系统类型都可以挂载。本章后续内容中会提供关于mount命令的其他用法。

^① cdrom类型的文件系统可以被非根用户挂载。

代码清单9-3显示了一个嵌入式系统配置的Flash设备中的目录内容。

代码清单9-3 Flash设备列表

```
$ ls -l /mnt/flash
total 24
drwxr-xr-x 2 root root 1024 Jul 18 20:18 bin
drwxr-xr-x 2 root root 1024 Jul 18 20:18 boot
drwxr-xr-x 2 root root 1024 Jul 18 20:18 dev
drwxr-xr-x 2 root root 1024 Jul 18 20:18 etc
drwxr-xr-x 2 root root 1024 Jul 18 20:18 home
drwxr-xr-x 2 root root 1024 Jul 18 20:18 lib
drwx----- 2 root root 12288 Jul 17 13:02 lost+found
drwxr-xr-x 2 root root 1024 Jul 18 20:18 proc
drwxr-xr-x 2 root root 1024 Jul 18 20:18 root
drwxr-xr-x 2 root root 1024 Jul 18 20:18/sbin
drwxr-xr-x 2 root root 1024 Jul 18 20:18 tmp
drwxr-xr-x 2 root root 1024 Jul 18 20:18 usr
drwxr-xr-x 2 root root 1024 Jul 18 20:18 var
$
```

代码清单9-3是一个嵌入式系统根文件系统中顶层（根）目录的例子。第6章就如何确定根文件系统的内容提供了指导和示例。

9.2.2 文件系统完整性检查

e2fsck命令用来检查一个ext2类型文件系统的完整性。由于某些原因，文件系统可能会被损坏，最常见的原因可能是不可预知的电源掉电、没有关闭所有打开的文件且没有卸载文件系统就关闭内部电源。Linux发行版恰好是在系统关机（假定按照正确顺序关闭系统）期间进行这些关闭文件的操作。然而，在处理嵌入式系统时，不可预知的电源掉电现象在嵌入式系统应用中是很常见的，因此，我们需要采取一些应对措施。e2fsck命令是对于采用ext2文件系统设备非正常关闭电源导致系统损坏问题的第一个防御办法。

在前面提到的CompactFlash设备上运行e2fsck命令后，其结果如代码清单9-4所示。由于格式化和卸载操作都正确，所以输出内容中没有错误提示。

代码清单9-4 正常文件系统检测

```
# e2fsck /dev/sdb1
e2fsck 1.37 (21-Mar-2005)
CFlash_Boot_Vol: clean, 23/2880 files, 483/11504 blocks
#
```

e2fsck命令会从多个方面检查文件系统的连续性。如果没有发现问题，执行e2fsck命令后会给出类似代码清单9-4中的信息。需要注意的是，e2fsck命令应该是在一个未挂载的文件系统上运行。尽管它可以运行在一个已挂载的文件系统上，但是这样做可能对磁盘或闪存设备上内部文件系统结构产生重大破坏。

再举一个更有趣的例子，代码清单9-5的内容是在CompactFlash设备还未卸载就从插口处拔掉

后执行e2fsck命令的结果。我们有意创建了一个文件并且使它在从系统移除之前处于编辑中，这样做的结果就会导致文件数据结构的损坏，也会损坏包含该文件数据的数据块。

代码清单9-5 异常文件系统检测

```
# e2fsck -y /dev/sdb1
e2fsck 1.37 (21-Mar-2005)
/dev/sdb1 was not cleanly unmounted, check forced.
Pass 1: Checking inodes, blocks, and sizes
Inode 13, i_blocks is 16, should be 8. Fix? yes

Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information

/dev/sdb1: ***** FILE SYSTEM WAS MODIFIED *****
/dev/sdb1: 25/2880 files (4.0% non-contiguous), 488/11504 blocks
#
```

从代码清单9-5可以看出，采用e2fsck命令可以检测到CompactFlash设备没有被正常卸载。此外，还可以看到e2fsck命令执行期间在文件系统上的处理过程。e2fsck命令会对整个文件系统进行5个阶段的检测，检测内容包括内部文件系统数据结构的组成部分。这里检测到一个被认为是节点（inode）^①13的文件错误，由于在e2fsck命令的命令行中带有-y选项，所以该错误会被自动修复。

当然，在一个实际系统中，可能就没有这么幸运了。对于某些文件系统错误类型来说，使用e2fsck命令是不能修复的。另外，嵌入式系统的设计者需要明白，如果没有按照正常的方式关闭电源，由于要扫描用户的引导设备并修复错误，系统的引导过程会延迟进行。实际上，如果这些错误不能修复，系统引导会被挂起，而此时就需要手动干涉。此外还要注意的，如果用户的文件系统十分庞大，文件系统检测过程（fsck）可能会花费几分钟甚至几个小时。

防止文件系统损坏的另一个措施，是确保磁盘写操作得以立即执行。系统同步机制（sync）可以强制所有I/O请求队列都能在相应的设备上响应。对于由于电源掉电或驱动失败带来的数据异常的缺陷，Windows采用在每次文件写操作后执行一个sync命令，或其他满足用户特定需要的策略，来尽可能减少这种影响，当然，这需要付出降低系统性能的代价。所有的现代操作系统都将延迟磁盘写操作作为性能优化的措施。有效使用同步机制的效果要优于这种优化措施。

作为Linux系统下已经成熟的ext2文件系统，它具有响应快速、高效、健壮的特点。但是，如果用户需要一个日志文件系统所具备的可靠性，或者用户设计中需要确定在非正常关机后的引导时间，就可以考虑采用ext3文件系统。

9.3 ext3 文件系统

ext3文件系统已经成为了一个功能强大、高性能并且健壮的日志文件系统。目前它已成为许

^① 在文件系统中，用ext2内部数据结构表示的一个文件称为一个节点。

多主流桌面Linux（诸如Red Hat、Fedora Core系列版本）的默认文件系统。

ext3文件系统是对ext2文件系统的扩展，主要增加了日志功能。所谓日志，是指对文件系统的每一个变化进行记录，从而可以从日志记录恢复文件系统的一种技术。ext3文件系统的最大优势，是在系统非正常关闭后可以直接被挂载。前一节曾提到，当一个系统被意外关闭，例如由于系统电源掉电而引起的系统关闭，系统会强制文件系统进行连续性检查，而这会是一个非常耗时的操作。如果是ext3文件系统，就不需要进行文件系统连续性检查，因为系统日志可以恢复系统意外关闭之前的状态，从而确保了文件系统的连续性。

这里不准备详细描述日志文件系统是如何工作的，因为这不属于本书内容，在此只做一些简要说明。日志文件系统包含一个由用户隐藏的特殊文件，该文件用来存储文件系统的元数据^①和文件数据，这个特殊的文件就是日志。当文件系统有变化（例如写操作）时，会首先将该变化记录到日志中，在文件系统发生变化之前，文件系统的使用者要确保将该变化写入日志并且保存到系统存储介质（如磁盘或者闪存）上。当文件系统的变化写入日志后，文件系统的使用者才会改变相应的文件和存储介质中的元数据。在将元数据写入到介质过程中，如果出现电源掉电并且在掉电之后重启了系统，那么要恢复文件系统的连续性，只需要重现记录在日志中的变化即可。

设计ext3文件系统最重要的一个目的，是可以向前或向后都兼容ext2文件系统。不用重新格式化或重写磁盘上的所有数据就可以实现ext2文件系统和ext3文件系统的相互转换，代码清单9-6详细描述了这一过程^②。

代码清单9-6 ext2文件系统转换为ext3文件系统

```
# mount /dev/sdb1 /mnt/flash <<< Mount the ext2 file system
# tune2fs -j /dev/sdb1 <<< Create the journal
tune2fs 1.37 (21-Mar-2005)
Creating journal inode: done
This filesystem will be automatically checked every 23 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
#
```

从代码清单9-6可以看到，为了举例说明，我们首先将文件系统挂载到/mnt/flash目录下，通常应在一个未被挂载的ext2类型分区上执行该操作。在将文件系统挂载之后执行tune2fs命令会生成名为.journal的日志文件，这是一个隐藏文件。Linux中以（.）开头的文件被系统视为隐藏文件。大多数命令行文件操作命令都会忽视这样的隐藏文件，通过代码清单9-7可以看到，执行带有-a选项的ls命令，就可以列出所有的文件，包括隐藏文件。

代码清单9-7 ext3 日志文件

```
$ ls -al /mnt/flash
total 1063
drwxr-xr-x 15 root root 1024 Aug 25 19:25
```

① 元数据是文件组成的基本单元，与文件数据相对应，包含文件创建日期、时间、大小、块使用信息等。

② 这样转换文件系统应该仅视作开发行为。

```

drwxrwxrwx  5 root root    4096 Jul 18 19:49 ..
drwxr-xr-x  2 root root    1024 Aug 14 11:27 bin
drwxr-xr-x  2 root root    1024 Aug 14 11:27 boot
drwxr-xr-x  2 root root    1024 Aug 14 11:27 dev
drwxr-xr-x  2 root root    1024 Aug 14 11:27 etc
drwxr-xr-x  2 root root    1024 Aug 14 11:27 home
-rw-----  1 root root 1048576 Aug 25 19:25 .journal
drwxr-xr-x  2 root root    1024 Aug 14 11:27 lib
drwx-----  2 root root   12288 Aug 14 11:27 lost+found
drwxr-xr-x  2 root root    1024 Aug 14 11:27 proc
drwxr-xr-x  2 root root    1024 Aug 14 11:27 root
drwxr-xr-x  2 root root    1024 Aug 14 11:27/sbin
drwxr-xr-x  2 root root    1024 Aug 14 11:27 tmp
drwxr-xr-x  2 root root    1024 Aug 14 11:27 usr
drwxr-xr-x  2 root root    1024 Aug 14 11:27 var

```

现在我们已经在一个Flash模块上创建了日志文件，它被有效地格式化为ext3文件系统。在再次引导系统或者在这个刚创建的ext3文件系统的分区上执行e2fsck时，日志文件就会自动隐藏，日志文件的元数据保存在一个为此而保留的inode集合中。如果用户可以看到这个.journal隐藏文件，那么修改或删除这个文件将会非常危险。

在不同存储设备上创建日志文件有时也可能是有益的。如果在用户的系统中有多个物理存储设备，可以把ext3类型的日志文件系统放到第一个驱动器上，把日志文件放到第二个驱动器中。不管用户的物理设备是基于闪存还是磁盘类的存储设备，都可以采用上述的方法。在一个独立分区上将一个具有日志文件的ext2文件系统转换为一个日志文件系统，可以通过调用tune2fs实现，其方法如下：

```
# tune2fs -J device=/dev/sda1 -j /dev/sdb1
```

如果要实现上述操作，用户必须要有一个已格式化的设备，并且该设备上的日志文件必须是ext3文件系统类型。

9.4 ReiserFS 文件系统

ReiserFS文件系统在一些桌面Linux版本（如SuSE和Gentoo）中得到了广泛应用。在写本书时，Reiser4仍是目前ReiserFS日志文件系统的代表。类似ext3文件系统，ReiserFS可以确保任何一个给定文件系统的操作，而不论该操作过程完成与否。但是与ext3文件系统不同的是，Reiser4会给系统程序员提供一个API，以保证文件系统事务的原子性。

当一个数据库程序正在忙于更新数据库中的记录时，文件系统正在发布一些写操作。假定在第一次写操作结束而最后一次写操作还没有完成时电源掉电，那么一个日志文件系统就可以确保将元数据的变化写入日志文件，这样当系统电源重新加电后，内核至少可以确保文件系统处于连续状态。更确切地说，如果文件A在电源掉电之前的大小为16KB，那么在加电之后仍会报告为16KB，而且目录列表中仍然会正确地列出这个文件（事实上是节点）大小的记录。然而，这并不意味着文件数据的正常写入，它只是提示此时文件系统没有错误。事实上，它很有可能在前面的环节中由于数据库程序而丢失了数据，而且它会直到有数据库逻辑的时候才恢复丢失的数据，

假定有恢复执行的话。

Reiser4实现了高性能的原子(atomic)文件系统操作,这是为了保护文件系统状态(连续性)和文件系统操作中所调用的数据。Reiser4提供了一个用户级的API,用来保证命令执行成功或失败时数据的完整性,例如数据库管理员发出的文件系统写程序。这不仅保证所维护的文件系统的连续性,同时也保证了系统崩溃后没有残留不完整的数据或废弃的数据。

关于ReiserFS的更多详细介绍,可以访问本章“参考资源”中提到的ReiserFS的主页。

9.5 JFFS2 文件系统

闪存已经在嵌入式产品中得到了广泛应用。闪存由于其固有技术特性,因此效率不高且在大量的写操作中由于电源掉电可能会造成数据损坏。块的大小是造成闪存低效的原由,其块大小通常都是几十千兆甚至几百千兆,一次必须擦除一个块,尽管有时一次只进行1字节或1个字的写操作,也必须先整块擦除然后再重写。

众所周知,在任何版本的Linux(或其他操作系统)中体积小文件要远远多于体积较大的文件,采用gnuplot所绘制的柱状图9-2显示了一个典型Linux系统下的文件大小分布。

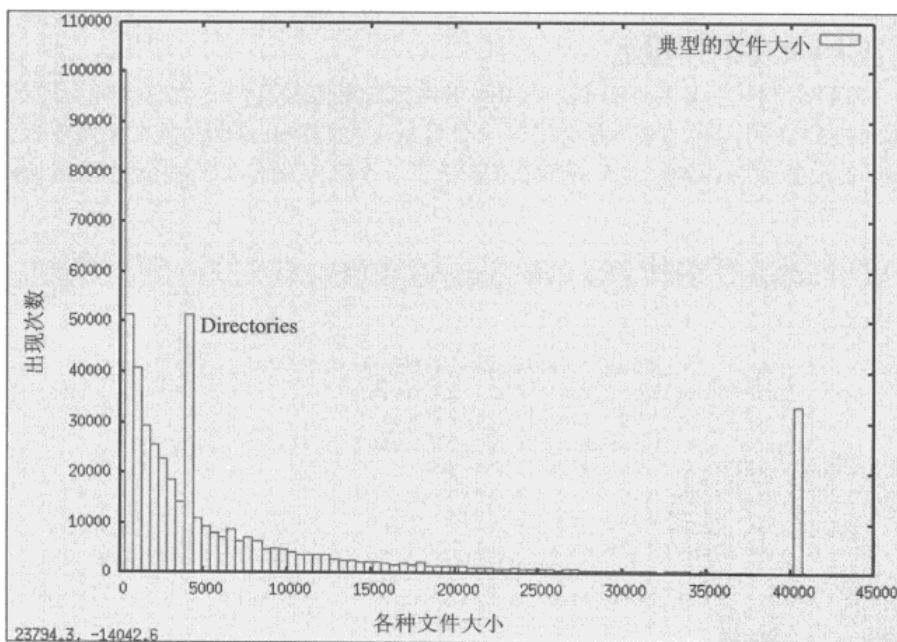


图9-2 文件大小分布(单位:字节)

从图9-2可以看到,许多文件的大小大约在10KB以下,4 096处的尖峰波形代表的是目录,目录项(也是所有的文件)恰好有4 096字节,该目录下有许多这样的文件。坐标40 000处以上的波形是人为测量的结果,可以看到只有几个文件的大小大于40KB。有趣的是,可以看到多数文件都很小。

大量的小体积文件给闪存文件系统的设计者带来了特殊的挑战，因为闪存一次必须整块擦除，且闪存块通常是这些小文件的大小的很多倍。闪存的块重写操作很耗时，例如，块大小为128KB的闪存用来存储大量大小为4KB或更小的文件。现在假定其中的一个文件需要修改，闪存文件系统将使整个128KB的块失效，并且将该闪存块中所有文件重新写到另一个新擦除的块中，这将是一个耗时的过程。

由于闪存的擦写周期较长（要比硬盘写操作慢许多），这就会增加在电源突然掉电时造成数据损坏的概率。例如，如果前面提到的闪存在重写128KB的数据块时突然掉电，那么该数据块中的所有文件都有可能丢失。

现在我们讨论JFFS2。通过设计第二代日志闪存文件系统（或称为JFFS2），已经在很大程度上减少或排除了所存在的上述文件系统的问题。JFFS文件系统最初由瑞典Axis通信公司（Axis Communications AB）开发，当时的目标是在普通闪存上使用。JFFS由于闪存的体系结构以及这种结构的缺陷性而为人们所熟知。

闪存上的文件系统存在的另一个问题是有限的使用寿命，典型的闪存具有最小10万次的擦写周期，现在更多的是具有100万次的擦写周期，每个闪存块都具有这样的特点。这种与众不同的特性要求闪存尽可能地使擦写操作在闪存上均匀分布。JFFS2采用一种称为磨损平衡（wear leveling）的技术来实现这样的功能。

构建一个JFFS2文件系统相对简单，必须确保内核总是支持JFFS2，且在你的开发平台中需要有一个mkfs.jffs2实用程序的兼容版本。在一个目录下构建的JFFS2映像文件需要该目录具备一个文件系统所要求的文件内容，代码清单9-8显示了一个闪存设备作为根文件系统使用时的典型目录结构。

代码清单9-8 JFFS2 文件系统的目录列表

```
$ ls -l
total 44
drwxr-xr-x 2 root root 4096 Aug 14 11:27 bin
drwxr-xr-x 2 root root 4096 Aug 14 11:27 dev
drwxr-xr-x 2 root root 4096 Aug 14 11:27 etc
drwxr-xr-x 2 root root 4096 Aug 14 11:27 home
drwxr-xr-x 2 root root 4096 Aug 14 11:27 lib
drwxr-xr-x 2 root root 4096 Aug 14 11:27 proc
drwxr-xr-x 2 root root 4096 Aug 14 11:27 root
drwxr-xr-x 2 root root 4096 Aug 14 11:27 sbin
drwxr-xr-x 2 root root 4096 Aug 14 11:27 tmp
drwxr-xr-x 2 root root 4096 Aug 14 11:27 usr
drwxr-xr-x 2 root root 4096 Aug 14 11:27 var
$
```

正确生成这个目录后，该目录下所列内容可以用作是mkfs.jffs2命令生成其映像文件的一个模板。使用mkfs.jffs2命令会在代码清单9-8所示的目录树中产生一个合适的JFFS2文件系统映像文件，通过mkfs.jffs2命令行参数可以指定操作目录的路径和所生成JFFS2映像的文件名。默认情况下是在当前路径下产生JFFS2映像文件。构建JFFS2映像文件的命令如代码清单9-9中所示。

代码清单9-9 mkfs.jffs2命令示例

```
# mkfs.jffs2 -d ./jffs2-image-dir -o jffs2.bin
# ls -l
total 4772
-rw-r--r-- 1 root root 1098640 Sep 17 22:03 jffs2.bin
drwxr-xr-x 13 root root 4096 Sep 17 22:02 jffs2-image-dir
#
```

代码清单9-8中的目录结构及其文件就是例子中的jffs2-image-dir目录。我们可以在这个文件系统映像的目录下执行mkfs.jffs2命令，通过该命令的-d参数指定文件系统模板的位置，使用-o参数命名使用mkfs.jffs2命令所生成的JFFS2映像文件。最终生成的映像文件jffs2.bin，将在第10章中研究MTD子系统和JFFS2文件时用到。

需要指出的是，任何支持写操作的基于闪存的文件系统都会受到这种情况的影响，即闪存设备的过早失效。例如，系统日志（syslogd和klogd）将数据写到基于闪存的文件系统会很容易导致对闪存设备不停地擦写。某些类型的程序错误也有可能对闪存的不停擦写，所以必须注意限制闪存的擦写次数，以确保闪存设备的使用寿命。

9.6 cramfs 文件系统

从cramfs项目工程中的README文件可以看到，cramfs文件系统的设计初衷就是在一个小容量的ROM（只读存储器）中填满文件系统。对于具有只存放静态数据或程序的小容量ROM或闪存的嵌入式系统来说，非常适合使用cramfs文件系统。这里再次借用cramfs里README文件中对cramfs的描述：“cramfs设计简洁、小巧，压缩得很好。”

cramfs文件系统是只读的，采用一个名为mkcramfs的命令行实用程序就可以生成cramfs文件系统。如果用户的开发平台中没有该实用程序，可以通过在本章末尾的相关链接下载该实用程序。类似于JFFS2，mkcramfs实用程序会使用特定命令在一个指定目录中构建cramfs文件系统映像文件。代码清单9-10详细描述了构建cramfs文件系统映像的过程。我们可以使用代码清单9-8中构建JFFS2映像文件的同一个文件系统结构来生成cramfs文件系统映像。

代码清单9-10 mkcramfs命令示例

```
# mkcramfs
usage: mkcramfs [-h] [-v] [-b blksize] [-e edition] [-i file] [-n name]
dirname outfile
-h          print this help
-E          make all warnings errors (non-zero exit status)
-b blksize  use this blocksize, must equal page size
-e edition  set edition number (part of fsid)
-i file     insert a file image into the filesystem (requires >= 2.4.0)
-n name     set name of cramfs filesystem
-p          pad by 512 bytes for boot code
-s          sort directory entries (old option, ignored)
-v          be more verbose
-z          make explicit holes (requires >= 2.3.39)
dirname    root of the directory tree to be compressed
```



```

outfile    output file

#
# mkcramfs . ../cramfs.image
warning: gids truncated to 8 bits (this may be a security concern)
# ls -l ../cramfs.image
-rw-rw-r-- 1 chris chris 1019904 Sep 19 18:06 ../cramfs.image

```

为了了解mkcramfs命令使用信息，可以不带任何命令行参数执行mkcramfs命令。由于mkcramfs实用程序没有帮助手册，所以这是理解该实用程序如何使用的最好方法。随后，我们在当前指定目录（cramfs文件系统文件所在的目录）下执行，指定目标文件名为cramfs.image。最后，列出了刚刚生成的文件，可以看到一个名为cramfs.image的新文件。

要注意的是，如果用户的内核配置为支持cramfs文件系统，就可以把这个文件系统映像挂载到用户的Linux开发平台中，同时可以查看该文件系统的内容。当然，由于cramfs文件系统是只读的文件系统，所以它的内容不能修改。代码清单9-11显示了将cramfs文件系统挂载到/mnt/flash目录下的操作过程。

代码清单9-11 查看cramfs文件系统

```

# mount -o loop cramfs.image /mnt/flash
# ls -l /mnt/flash
total 6
drwxr-xr-x 1 root root 704 Dec 31 1969 bin
drwxr-xr-x 1 root root 0 Dec 31 1969 dev
drwxr-xr-x 1 root root 416 Dec 31 1969 etc
drwxr-xr-x 1 root root 0 Dec 31 1969 home
drwxr-xr-x 1 root root 172 Dec 31 1969 lib
drwxr-xr-x 1 root root 0 Dec 31 1969 proc
drws----- 1 root root 0 Dec 31 1969 root
drwxr-xr-x 1 root root 272 Dec 31 1969 sbin
drwxrwxrwt 1 root root 0 Dec 31 1969 tmp
drwxr-xr-x 1 root root 124 Dec 31 1969 usr
drwxr-xr-x 1 root root 212 Dec 31 1969 var
#

```

你或许已注意到了，当执行mkcramfs命令后会产生关于用户组ID（GID）的警告信息。为了缩减文件系统大小和加速系统执行，cramfs文件系统采用了非常简洁的元数据。cramfs文件系统的特性是它只保留了GID的低8位，而Linux采用的是16位的GID，这样用大于255的GID生成的文件就会有所删减，即产生代码清单9-10中提示的警告信息。

尽管cramfs文件系统在文件大小、文件最大数量等方面有限制，但是cramfs文件系统对于采用ROMS引导的系统（只读操作，快速压缩的方式）来说是非常理想的。

9.7 NFS 文件系统

已经在UNIX环境下做过开发的读者毫无疑问会对NFS（网络文件系统）比较熟悉。如果正确地配置了NFS文件系统，用户就可以将一个目录输出到NFS服务器上，并可以将该目录挂载到一个远程客户端机器上。对客户端的使用者来说，挂载的该目录就好像是本机的一个文件系统一

样。对于采用大型网络操作系统（如Unix/Linux）的机器来说，采用NFS服务是非常有意义的，对于嵌入式开发者来说更是如此。如果用户的开发板上启用了NFS服务，即使用户使用一个资源十分有限的嵌入式系统，也可以在开发过程中获取大量文件、函数库、工具及各种程序。

类似于其他文件系统，服务器端和客户端的用户的内核都必须配置为支持NFS。不过，对于服务器和客户端来说，在内核中配置NFS的功能是相互独立的。

配置和使用NFS的详细说明不在本书描述范围内，但是简短的介绍有助于说明在嵌入式系统中是如何使用NFS的。本章最后的“参考资源”中会有NFS相关的链接，其中包括完整的NFS-Howto文档。

在一个支持NFS功能的开发平台上，每一个想要通过NFS输出的目录名都会包含在一个文件中。在Red Hat或其他版本下，这个名为exports的文件位于/etc目录下。代码清单9-12显示的是一个/etc/exports文件内容，该文件或许在用于嵌入式开发的开发平台下能够找到。

代码清单9-12 /etc/exports文件内容

```
$ cat /etc/exports
# /etc/exports
/home/chris/sandbox/coyote-target *(rw,sync,no_root_squash)
/home/chris/workspace *(rw,sync,no_root_squash)
$
```

代码清单9-12中的/etc/exports文件包含了在一个Linux开发平台下的两个目录名。第一个目录包含了ADI Engineering Coyote评估板中的一个目标文件系统，第二个目录是嵌入式系统目标项目下的一个普通工作目录，该目录的设置可以是任意的，用户可以按照自己的选择进行设置。

在一个启用了NFS服务的嵌入式系统中，可以使用下面的命令将通过NFS服务器输出的.../workspace目录挂载到我们选择的挂载点上：

```
$ mount -t nfs pluto:/home/chris/workspace /workspace
```

注意这个命令的几个要点。在该命令中，我们通知mount命令将一个远程目录（位于名为pluto的机器中）挂载到本地的挂载点/workspace下。为了让该命令正确执行，嵌入式目标系统必须满足两个要求。首先，目标系统必须能够识别名为pluto的机器名，而且必须能够解析该机器名。最简单的方法是在目标系统下的/etc/hosts文件中添加一个入口，这将使得网络子系统的机器名和其IP地址相对应。在该目标系统下/etc/hosts文件中添加的入口如下：

```
192.168.10.9      pluto
```

第二个要求是在该嵌入式目标系统的/root目录下有称为/workspace的目录，该目录即是挂载点。这样执行前面提到的挂载命令后，就可以将NFS服务器上的/home/chris/workspace目录挂载到嵌入式系统的/workspace下。

上面的操作非常有用，特别是在交叉开发环境中更是如此。当用户正在开发一个非常大的项目时，每次针对项目的修改，用户都需要将应用程序下载到目标板中进行测试和调试。如果像前面所描述那样采用NFS工作模式，假定用户正处于主机（host）中采用NFS输出的目录下，那么在用户的嵌入式开发系统下会立即得到相应的更新，而不需要重新上传编译后的工程文件，这样

可以极大地加快开发过程。

采用 NFS 挂载根文件系统

将用户的工作空间挂载到嵌入式目标板上，将对目标板的开发和调试非常有利，尤其是在源码级的调试中，可以快速获取用户空间中的源代码及其相应的改变。当目标开发系统的资源严格受限时，这一优点则更加明显。当用户想要从NFS服务器上将一个嵌入式系统的根文件系统全部挂载时，采用NFS将是一个非常好的工具。注意代码清单9-12中的coyote-target开发板挂载到开发平台下的根文件系统目录，该目录包括了成千上万个与目标体系结构兼容的文件。

针对嵌入式系统的主流嵌入式Linux版本具有成千上万个编译文件，并且在一些选择的目标体系结构下经过了测试。为此，在代码清单9-13中，我们列出了在代码清单9-12中提到的coyote-target开发板下的目录清单。

代码清单9-13 目标文件系统示例

```
$ du -h --max-depth=1
724M    ./usr
4.0K    ./opt
39M     ./lib
12K     ./dev
27M     ./var
4.0K    ./tmp
3.6M    ./boot
4.0K    ./workspace
1.8M    ./etc
4.0K    ./home
4.0K    ./mnt
8.0K    ./root
29M     ./bin
32M     ./sbin
4.0K    ./proc
64K     ./share
855M    .
$
$ find -type f | wc -l
29430
```

从上面代码清单可以看到，该目标文件系统包括了几十亿字节的针对ARM体系结构的二进制文件。该目录下包括了29 000多个文件，包括二进制程序、配置文件和文本文件，这很难装在一个普通嵌入式系统的闪存设备中。

这就是通过NFS挂载一个根目录的原因。从开发目的考虑，如果将用户的嵌入式系统加载了在Linux工作站上所有用户熟悉的工具和程序，那么可以提高开发效率。事实上，很可能几十个不曾见过的命令行工具和开发程序就有助于用户缩短开发时间。第13章将介绍更多这些有用的工具。

要让用户的嵌入式系统在启动时就能通过NFS挂载其根文件系统，相对比较简单。首先，必须配置目标平台的内核可以支持NFS服务，在内核配置中也有一个选项允许用户通过NFS进行远

程根目录挂载。该配置选项如图9-3所示。

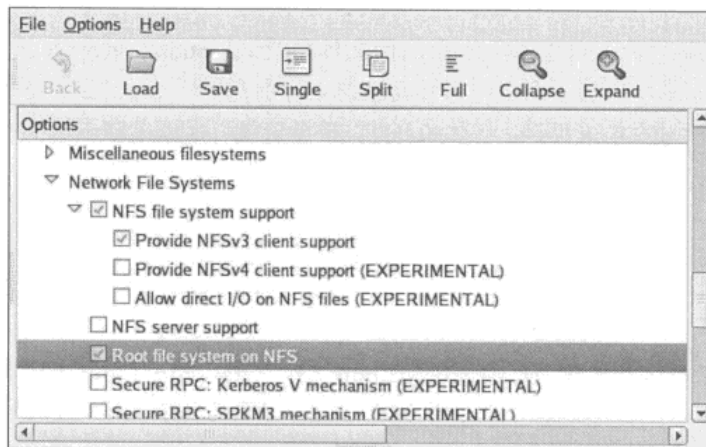


图9-3 NFS内核配置

从图9-3中可以看到已经选择了支持NFS文件系统，同时还选择了“Root file system on NFS”。在内核配置过程中选择了这些配置项之后，其余的工作就是给内核传递一些信息，以确保内核知道在什么地方可以访问NFS服务器。这可以通过一些方法来实现，其中一些要依赖于所选的目标体系结构和所选的引导装入程序。就一个最小系统来说，可以在系统加电时通过命令行来配置IP端口号和服务器，并将这些正确的参数传递给内核，方法如下：

```
console=ttyS0,115200 ip=bootp root=/dev/nfs
```

该命令告诉内核，希望通过NFS服务得到一个根文件系统，同时通过该命令也获得BOOTP服务器的一些相关信息（服务器名、服务器IP地址和根目录的挂载点）。在开发项目时，这是一个普通但十分有用的配置命令。如果用户采用静态方式配置目标板的IP地址，相应的内核命令行如下所示：

```
console=ttyS0,115200 \
ip=192.168.1.139:192.168.1.1:192.168.1.1:255.255.255.0:coyote1:eth0:off \
nfsroot=192.168.1.1:/home/chris/sandbox/coyote-target \
root=/dev/nfs
```

当然，上面的命令可以放在一行中，其中的ip=参数是在.../net/ipv4/目录下ipconfig.c文件中定义的，其语法如下（采用一行显示）：

```
ip=<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<device>:<PROTO>
```

在该语句中，client-ip是目标板的IP地址，server-ip是NFS服务器的IP地址，gw-ip是路由器网关的IP地址。如果服务器的IP和目标板的IP属于不同的网段，子网掩码netmask会定义IP地址类型。参数hostname是要传递的目标板主机名，device是Linux网络设备名，如eth0。参数PROTO定义了获取初始化IP参数所用的协议。

9.8 伪文件系统

在Linux内核配置菜单下可以看到大量的伪文件系统（Pseudo File System），这些伪文件系统为许多应用提供了大量有用的工具。作为补充信息，特别是对于proc文件系统，可以花一个下午的时间来研究这个有用的系统工具。本章最后的“参考资源”有该文件系统的一些参考阅读信息。

9.8.1 proc 文件系统

/proc文件系统的名字来源于它最初的使用目的，即为Linux系统内核和每一个运行进程提供通信接口，在这期间可以提供更多的进程信息。这里将着重介绍/proc文件系统，完整的/proc文件系统体验将作为练习留给读者。

/proc文件系统事实上已成为，除最简单Linux系统之外，所有Linux系统都具备的组成部分，即使是嵌入式的Linux系统也如此。许多用户级功能的实现都要依赖于/proc文件系统的内容。例如，不带任何命令行参数执行mount命令，执行该命令后会通过/proc/mounts文件提供的信息列举出当前运行系统下所有正在使用的挂载点。如果没有/proc文件系统可用，执行该命令将不会返回任何信息。上述操作在ADI Engineering Coyote开发板上的执行过程如代码清单9-14所示。

代码清单9-14 挂载依赖于/proc

```
# mount
rootfs on / type rootfs (rw)
/dev/root on / type nfs
(rw,v2,rsize=4096,wsiz=4096,hard,udp,nolock,addr=192.168.1.19)
tmpfs on /dev/shm type tmpfs (rw)
/proc on /proc type proc (rw,nodiratime)

< Now unmount proc and try again ...>

# umount /proc
# mount
#
```

从代码清单9-14可以注意到，/proc本身就是作为一个已挂载的文件系统列出的，其类型为proc，挂载点为/proc。用户必须在根目录树下有一个名为/proc的目录作为/proc文件系统的挂载点^①。要挂载一个/proc文件系统，可以使用与挂载其他文件系统类似的mount命令。

```
$ mount -t proc /proc /proc
```

由mount命令的帮助文档可知，执行mount命令的一般格式为mount fstype something somewhere，我们用none来代替设备名称（/proc），该命令如下：

```
$ mount -t proc none /proc
```

① 毫无疑问，可以将/proc文件系统挂载到文件系统下用户任意选择的地方，但是所有工具（包括mount）都希望将proc文件系统挂载到/proc。

该命令看起来有些故弄玄虚，something参数并不是严格必需的，因为/proc文件系统是一个虚拟文件系统而不是一个真实的物理设备。但是在前一个mount命令中指定了参数/proc，是在提醒我们将/proc文件系统挂载到了/proc目录下（更确切地说是/proc挂载点）。

显然，通过这样的操作，就可以使用/proc文件系统的功能了，但前提必须在内核的配置中选择使用/proc文件系统。该配置选项可以在伪文件系统下的文件系统子菜单中找到。

内核中运行的每一个用户进程都会有/proc文件系统下的一个目录对应。例如，第6章介绍的初始化进程id（PID）总是被分配为1。/proc中的进程就会有一个用它的PID命名的目录相对应。例如，PID为1的初始化进程会在/proc下有一个/proc/1目录与之对应。在Coyote嵌入式开发板下的显示内容如代码清单9-15所示。

代码清单9-15 /proc目录下所对应的初始化进程

```
# ls -l /proc/1
total 0
-r----- 1 root root 0 Jan 1 00:25 auxv
-r--r--r-- 1 root root 0 Jan 1 00:21 cmdline
lrwxrwxrwx 1 root root 0 Jan 1 00:25 cwd -> /
-r----- 1 root root 0 Jan 1 00:25 environ
lrwxrwxrwx 1 root root 0 Jan 1 00:25 exe -> /sbin/init
dr-x----- 2 root root 0 Jan 1 00:25 fd
-r--r--r-- 1 root root 0 Jan 1 00:25 maps
-rw----- 1 root root 0 Jan 1 00:25 mem
-r--r--r-- 1 root root 0 Jan 1 00:25 mounts
-rw-r--r-- 1 root root 0 Jan 1 00:25 oom_adj
-r--r--r-- 1 root root 0 Jan 1 00:25 oom_score
lrwxrwxrwx 1 root root 0 Jan 1 00:25 root -> /
-r--r--r-- 1 root root 0 Jan 1 00:21 stat
-r--r--r-- 1 root root 0 Jan 1 00:25 statm
-r--r--r-- 1 root root 0 Jan 1 00:21 status
dr-xr-xr-x 3 root root 0 Jan 1 00:25 task
-r--r--r-- 1 root root 0 Jan 1 00:25 wchan
```

在/proc文件系统下的每一项内容都代表着一个运行中的进程，此外还有一些更有用的信息，尤其是分析和调试进程的信息。例如，cmdline包含了启动进程时所调用的命令行，包括任何参数。cwd和root目录包含了当前工作目录的进程信息和当前的根目录。

/proc下对于系统调试非常有用的一个文件是maps，该文件列出了为程序分配的每一段虚拟内存和相关特征信息。以init为例，/proc/1/maps的输出内容如代码清单9-16所示。

代码清单9-16 /proc下init进程内存分布

```
# cat /proc/1/maps
00008000-0000f000 r-xp 00000000 00:0a 9537567 /sbin/init
00016000-00017000 rw-p 00006000 00:0a 9537567 /sbin/init
00017000-0001b000 rwxp 00017000 00:00 0
40000000-40017000 r-xp 00000000 00:0a 9537183 /lib/ld-2.3.2.so
40017000-40018000 rw-p 40017000 00:00 0
4001f000-40020000 rw-p 00017000 00:0a 9537183 /lib/ld-2.3.2.so
40020000-40141000 r-xp 00000000 00:0a 9537518 /lib/libc-2.3.2.so
```

```

40141000-40148000 ---p 00121000 00:0a 9537518 /lib/libc-2.3.2.so
40148000-4014d000 rw-p 00120000 00:0a 9537518 /lib/libc-2.3.2.so
4014d000-4014f000 rw-p 4014d000 00:00 0
befeb000-bf000000 rwxp befeb000 00:00 0
#

```

很显然，这些信息是有用的。你可以在上表中最开始的两行看到init进程在内存中的段地址，也可以看到init进程所用的共享库在内存中分配的段地址，其描述格式如下：

```
vmstart-vmend attr pgoffset devname inode filename
```

这里的vmstart和vmend分别表示虚拟内存中的起始地址和结束地址，attr表示所在内存区域的特征，如读、写或执行，同时也表示了该段内存是否可共享。pgoffset表示的是内存区域的页偏移量（内核虚拟内存的一个参数）。参数devname，其格式为xx:xx；代表设备ID在内核中的声明。如果没有为一个文件或一个设备分配内存区域，则devname为00:00。代码清单9-14中最后的两项是与给定内存区域相关的节点和文件。当然，如果该区域中没有文件和节点，就显示为0，这种情况下通常是一些数据段。

每一个进程的其他有用描述信息也在/proc目录中列出。这些项包含了关于运行中的进程状态信息，包括父进程ID（PID）、用户ID和组ID、虚拟内存使用的进程状态、信号量和功效。本章最后会有更详细的参考资料。

/proc下经常用到的一些项有cpuinfo、meminfo和version。cpuinfo描述了运行系统的处理器（CPU）信息，meminfo提供了系统内存所有使用情况的统计，而version则反映了Linux内核版本的序列号，包括编译内核所用编译器和所用于构建内核的机器信息。

Linux内核在/proc目录中提供了许多有用的内容，我们只是对/proc这个子系统作了浅显的介绍。Linux目前已设计了许多实用程序来提取和报告包含在/proc文件系统上的信息。常见的两个例子top和ps，这是每个嵌入式Linux开发者都应该熟悉的实用程序，这两个实用程序将第13章中详细介绍。其他一些与/proc文件系统接口的有用实用程序包括free、pkill、pmap和uptime，这些实用程序的具体内容可参见procpss软件包。

9.8.2 sysfs 文件系统

像/proc文件系统一样，sysfs文件系统没有与一个实际的物理设备相关联。相反地，sysfs模拟了特定的内核目标，如实际的物理设备，同时提供了一种关联设备和设备驱动程序的方法。典型Linux版本的某些代理就要依赖于sysfs中的信息。

通过直接查看sysfs的目录结构，就可以知道哪些目标会输出到/sys下。代码清单9-17列出了Coyote开发板中/sys的顶层目录内容。

代码清单9-17 /sys顶层目录内容

```

# dir /sys
total 0
drwxr-xr-x 21 root root 0 Jan 1 00:00 block
drwxr-xr-x 6 root root 0 Jan 1 00:00 bus
drwxr-xr-x 10 root root 0 Jan 1 00:00 class

```

```
drwxr-xr-x  5 root    root    0 Jan  1 00:00 devices
drwxr-xr-x  2 root    root    0 Jan  1 00:00 firmware
drwxr-xr-x  2 root    root    0 Jan  1 00:00 kernel
drwxr-xr-x  5 root    root    0 Jan  1 00:00 module
drwxr-xr-x  2 root    root    0 Jan  1 00:00 power
#
```

可以看到，sysfs为每个系统主要设备（包括系统总线）提供了一个子目录。例如，在block子目录下，每一个块设备都作为一项列出，在/sys顶层目录下的其他子目录也都如此。

绝大多数由sysfs存储的信息会以更适合于机器阅读的格式表示。例如，为了发现在PCI总线上的设备，用户可以直接查看/sys/bus/pci子目录的信息。在我所用的Coyote开发板上有一个单独的PCI设备（以太网卡），查看其子目录信息如下：

```
# ls /sys/bus/pci/devices/
0000:00:0f.0 -> ../../../../devices/pci0000:00/0000:00:0f.0
```

该内容实际上显示的是指向sysfs目录树下另一个节点的符号链接，这里按照一定格式（保持在一行中）输出了该PCI设备信息。符号链接名是内核对PCI总线的描述，并且指向了称为pci0000:00（PCI总线描述）的devices子目录，它包括了大量的子目录和描述该PCI设备特性的文件。可以看到，这些数据很难发现和解析。

Linux有一个浏览sysfs文件系统目录结构的有效实用程序，称为systool。该实用程序可以从sourceforge.net网站上的sysutils软件包中获得。下面通过systool显示了前面讨论的PCI总线信息。

```
$ systool -b pci
Bus = "pci"
0000:00:0f.0 8086:1229
```

通过上面的命令，我们再次看到了内核对总线和总线设备（0f）的描述，但是这一次显示了从/sys下的/sys/devices/pci0000:00获得的制造商ID（8086=Intel）和设备ID（1229=cepro100以太网卡）。不带任何参数执行systool，之后会显示系统顶层目录结构。以Coyote开发板为例，其输出如代码清单9-18所示。

代码清单9-18 systool输出内容

```
$ systool
Supported sysfs buses:
    i2c
    ide
    pci
    platform
Supported sysfs classes:
    block
    i2c-adapter
    i2c-dev
    input
    mem
    misc
```

```

net
pci_bus
tty
Supported sysfs devices:
pci0000:00
platform
system

```

可以看到来自sysfs的各类系统信息，许多程序利用该信息来确定某些设备特征，或加强系统措施，例如电源管理和热插拔性能。

9.9 其他文件系统

Linux支持众多的文件系统，限于篇幅这里没有一一提及。然而，你应该知道一些其他嵌入式系统中经常用的文件系统。

从Linux源代码中关于ramfs模块文件的内容可知，ramfs文件系统是文件系统选择的最佳考虑。代码清单9-19列出了该文件中开始几行的内容。

代码清单9-19 Linux ramfs源代码模块内容

```

/*
 * Resizable simple ram filesystem for Linux.
 *
 * Copyright (C) 2000 Linus Torvalds.
 *           2000 Transmeta Corp.
 *
 * Usage limits added by David Gibson, Linuxcare Australia.
 * This file is released under the GPL.
 */

/*
 * NOTE! This filesystem is probably most useful
 * not as a real filesystem, but as an example of
 * how virtual filesystems can be written.
 *
 * It doesn't get much simpler than this. Consider
 * that this file implements the full semantics of
 * a POSIX-compliant read-write filesystem.

```

该模块主要作为如何对虚拟文件系统进行写操作的一个例子。ramfs文件系统与当今Linux内核中的ramdisk的最主要区别就是，ramfs文件系统可以在使用中改变大小，而ramdisk并不具备这样的能力。该源代码模块写得简洁高效，在这里列举出来可以作为学习之用，你可以仔细研究一下这个很好的例子。

tmpfs文件系统与ramfs类似，而且与ramfs相关。类似于ramfs，tmpfs中所有内容都保存在Linux内核的虚拟内存中，而且tmpfs的内容也会在系统掉电或系统重启时丢失。tmpfs文件系统有利于临时文件的存储。在一个音频应用方案中，我就使用挂载到/tmp目录下的tmpfs文件系统来加速音频子系统对临时文件的快速创建和删除。这也是在系统重启后保持/tmp目录被清空的绝好方法。挂载tmpfs文件系统与挂载其他虚拟文件系统的方法类似。


```
# mount -t tmpfs /tmpfs /tmp
```

就像其他虚拟文件系统（如/proc文件系统），前面的挂载命令中的第一个/tmpfs参数是“no-op”（空操作），它可以用单词none来表示，但仍然发挥着作用。这里使用/tmpfs参数是提示用户，即正在挂载一个称为tmpfs的文件系统。

9.10 构建简单的文件系统

构建简单文件系统的映像很容易，这里我们用Linux内核的回环设备（loopback device）来示范。Linux的回环设备可以把一个正式文件当作一个块设备来使用。简而言之，在构建文件系统映像之后用Linux的回环设备挂载该映像文件，其挂载方法与挂载其他块设备的方法一样。

要构建一个简单的根文件系统，首先要创建一个大小固定的全0文件，创建命令如下：

```
# dd if=/dev/zero of=./my-new-fs-image bs=1k count=512
```

该命令创建了一个大小为512KB的文件，文件内容全为0。用0来填充该文件有助于后面的文件压缩工作，并且可以保持文件系统中未初始化数据块数据格式的一致性。使用dd命令需要谨慎，如果不指定复制大小边界（count=）或者指定了错误的大小边界，执行dd命令就可能会将用户的硬盘填满数据块，甚至可能使用户系统崩溃。dd是一个功能强大的工具。dd等命令被敲错，再被根用户执行后，破坏了无数的文件系统。

当已经创建了一个新的映像文件之后，我们通常要格式化该文件，以建立一个指定文件系统定义的数据结构。这一详细过程如代码清单9-20所示。

代码清单9-20 创建一个ext2文件系统映像

```
# /sbin/mke2fs ./my-new-fs-image
mke2fs 1.35 (28-Feb-2004)
./my-new-fs-image is not a block special device.
Proceed anyway? (y,n) y
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
64 inodes, 512 blocks
25 blocks (4.88%) reserved for the super user
First data block=1
1 block group
8192 blocks per group, 8192 fragments per group
64 inodes per group

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 24 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
#
```

像dd命令一样，mke2fs命令也可能摧毁用户的系统，所以需要小心使用。在本例中，我们用mke2fs命令格式化一个文件而不是一个硬盘分区（块设备）。同样地，mke2fs会检测是否为块设备并且询问用户是否确认了该操作，在用户确认后，mke2fs会将一个ext2超级块（superblock）

和文件系统的数据结构写到文件中，然后我们可以使用Linux回环设备将该文件像一个块设备那样挂载，其挂载命令如下：

```
# mount -o loop ./my-new-fs-image /mnt/flash
```

该命令将文件my-new-fs-image作为一个文件系统挂载到/mnt/flash目录下，所用的挂载名并不重要，用户可以将该文件挂载到用户任意指定的地方，前提是只要该挂载点存在。也可以使用mkdir命令来创建用户的挂载点。

在将最新创建的映像文件作为一个文件系统挂载以后，我们就可以对该文件系统进行任意改动，可以增加或删减目录，可以创建设备节点，等等。我们也可以使用tar命令来复制或解压文件。当所有改动完成之后，这些改动保存到了文件之中，当然，前提是假定改动后的文件大小没有超过设备大小。需要记住的是，使用该种方法创建的文件系统，其大小在创建之初是固定的，不能改变。

9.11 小结

- 分区是一个物理设备的逻辑划分，Linux支持许多种分区类型。
- Linux下的文件系统挂载到了挂载点之上，根文件系统挂载到了文件系统的根目录下（以/表示）。
- 主流的ext2文件系统成熟而且高速，它经常用于嵌入式系统和其他一些Linux系统中，如Red Hat 和Fedora Core系列。
- ext3文件系统在ext2文件系统的基础上增加了日志管理，数据更加完整，系统也更加可靠。
- ReiserFS是另一个广受欢迎的高性能日志文件系统，也经常用于嵌入式和其他Linux系统中。
- 如果采用闪存，JFFS2是一个最优化的日志文件系统，它包含了友好使用闪存的一些特征，如采用磨损平衡机制以延长闪存使用寿命。
- cramfs是一个只读的文件系统，更适用于采用小容量引导ROM以及代码和数据只读的系统。
- NFS是嵌入式开发者所使用的功能最强大的开发工具之一，它使用户的目标开发系统具有工作站的强大功能。掌握如何将NFS作为用户的嵌入式目标板的根文件系统，这会给开发带来便利并且可以节省时间。
- Linux下有许多虚拟文件系统，这里介绍了其中几个重要的虚拟文件系统，如proc文件系统和sysfs文件系统。
- 基于RAM的tmpfs文件系统在嵌入式系统开发中有很多应用。相对于传统的ramdisk，tmpfs文件系统最大的改进是它可以动态调整自身大小以满足实际的操作需要。

参考资料

“Design and Implementation of the Second Extended Filesystem”

Rémy Card, Theodore Ts'o, and Stephen Tweedie

首次发表于*Proceedings of the First Dutch International Symposium on Linux*
<http://e2fsprogs.sourceforge.net/ext2intro.html>

“A Non-Technical Look Inside the EXT2 File System”

Randy Appleton

www.linuxgazette.com/issue21/ext2.html

白皮书: *Red Hat's New Journaling File System: ext3*

Michael K. Johnson

www.redhat.com/support/wpapers/redhat/ext3/

ReiserFS主页

www.namesys.com/

“JFFS: The Journaling Flash File System”

David Woodhouse

<http://sources.redhat.com/jffs2/jffs2.pdf>

cramfs 项目的README文件

Unsigned (assumed to be the project author)

<http://sourceforge.net/projects/cramfs/>

NFS主页

<http://nfs.sourceforge.net>

/proc文件系统文档

www.tldp.org/LDP/lkmpg/2.6/html/c712.htm

File System Performance: The Solaris OS, UFS, Linux ext3, and ReiserFS

技术白皮书

Dominic Kay

www.sun.com/software/whitepapers/solaris10/fs_performance.pdf



本章内容

- 启用MTD服务
- MTD基础知识
- MTD分区
- MTD实用程序
- 小结

为了支持大量闪存存储芯片之类的存储器设备，内存技术设备（MTD—The Memory Technology Devices）子系统应运而生。随着大量闪存编程方法的出现，我们可以使用很多不同类型的闪存芯片，这种情况产生的部分原因是由于有了许多专业而且高性能工作模式的支持。采用MTD层级的体系结构使得系统在使用存储器设备时可以将底层设备的复杂性同上层数据的组织和存储模式分离开来。

在这一章，我们将介绍MTD子系统，并通过一些简单的例子来介绍它的使用方法。首先我们将着眼于如何配置内核以支持MTD技术。我们将在一个支持MTD技术的开发平台上介绍一些简单的操作，以便对这个子系统有基本的认识。在这一章里，我们将把MTD技术同JFFS2文件系统结合在一起进行研究。

接着我们将介绍涉及MTD层的分区概念。我们将通过引导装入程序（bootloader）研究MTD分区建立过程中的技术细节，以及Linux内核是如何检测到分区的。接下来的内容将对MTD实用程序做简要介绍，最后总结所有内容，并启动一个具有JFFS2文件系统映像（存在于闪存中）的目标板。

10.1 启用 MTD 服务

为了使用MTD服务，用户必须对内核进行配置以启用MTD功能。在内核中有很多针对MTD的选项，其中有一些容易让人产生混淆。理解这些选项的最好方法是亲自配置一下。为了解释MTD子系统的使用机制以及它是如何配合系统工作的过程，我们从一些简单的例子开始介绍，这些例子读者可以在自己的Linux开发平台上实现。图10-1介绍了内核配置（由常用的make ARCH=<arch> gconfig调用实现），这个配置可以使系统具有最简单的MTD功能。代码清单 10-1

显示了进行图10-1所示的内核配置选择以后所生成的.config文件内容。

代码清单10-1 .config文件所示的MTD 基本配置

```
CONFIG_MTD=y
CONFIG_MTD_CHAR=y
CONFIG_MTD_BLOCK=y
CONFIG_MTD_MTDRAW=m
CONFIG_MTDRAW_TOTAL_SIZE=8192
CONFIG_MTDRAW_ERASE_SIZE=128
```

通过第一个配置选项来启用MTD子系统，该选项对应为图10-1中的第一个复选框（Memory Technology Device (MTD) Support）。对于在图10-1中接下来所选中的两个配置选项，这两个选项可以使用户层对闪存这样的MTD设备进行设备级的访问。第一个选项（CONFIG_MTD_CHAR）允许进行字符设备模式的访问，本质上讲就是那些一次一个字节连续读写的串行访问，而第二个选项（CONFIG_MTD_BLOCK）允许进行块设备模式的访问，这种访问方法用于磁盘驱动，在该种模式下会一次读写多个字节块的数据。这种访问方式允许用户使用熟悉的Linux命令来读写闪存中的数据，读者不久就会看到。

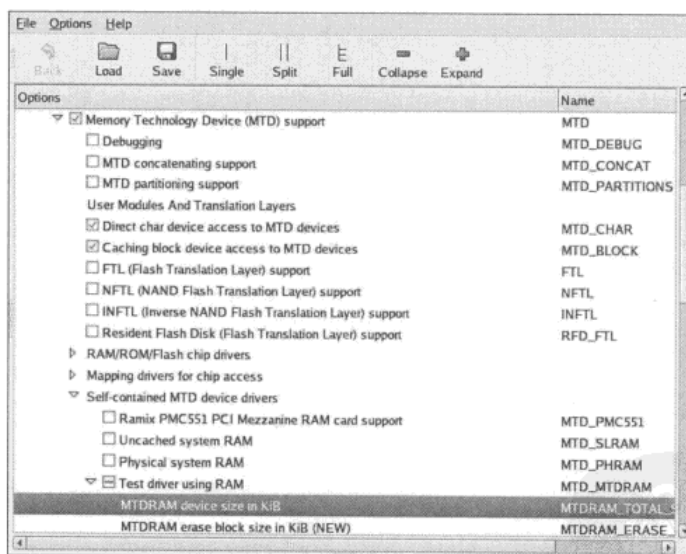


图10-1 MTD配置

CONFIG_MTD_MTDRAW配置会启用一个特殊的测试驱动程序，使得我们即使在没有任何MTD设备（如闪存）可用的情况下也能检测这个MTD子系统。与此配置选项相关的还有两个与RAM检测驱动程序有关的参数：设备大小和擦除大小。例如，这里我们指定了设备总大小为8192KB，擦除大小为128KB。这个测试驱动程序的作用是模拟闪存设备，首要目的是方便MTD子系统的测试与开发。因为闪存由固定大小的擦写块构成，测试驱动程序当然包括了擦写块的概念。读者马上会看到这些参数的具体用法。

构建 MTD

现在较新Linux内核的所有快照版本都包含了MTD功能。但是,如果读者想要利用一些MTD的新特性,而且这些新特性是读者所使用Linux内核版本发布后新增加的,那就必须下载并重建MTD驱动程序及其应用工具。由于MTD工具包既包含了内核组件也包含了用户空间的应用程序,那么就有必要把MTD工具包放在一个独立的工程目录下,并且把工具包和Linux内核源码树连接起来。在这里把MTD和用户的内核源码树集成起来的最简单的方法,是使用MTD工具包所提供的脚本。

读者可以从本章末尾给出的地址处下载MTD工具包,然后用tar命令将文件解压到指定目录。进入该目录并运行MTD工具包附带的patchkernel.sh脚本文件,该脚本文件提供了几个选项,在不带参数的情况下执行该脚本可以获得其详细用法,代码清单10-2中的内容显示了内核组件的安装方法。

代码清单10-2 为读者的内核加上MTD补丁

```
$ ./patchkernel.sh -2 ../sources/linux-2.6.10-MTD
Patching ../sources/linux-2.6.10-MTD/
Include JFFS2 file system: jffs2
Include JFFS3 file system (experimental): no
Method: ln      << Will actually create symbolic links
Can we start now ? [y/N]y

$
```

用参数-2调用patchkernel.sh脚本文件,表明我们想让系统支持JFFS2文件系统。在该命令中所提供的内核源码目录路径为../sources/linux-2.6.10-MTD/。默认情况下,patchkernel.sh脚本并不会将任何文复制到内核源码目录下,而是在内核源码树里创建指向MTD子目录的符号链接。这样一来,开发工作站上如有多个不同的内核,就可能共用一个MTD源码树。此后,读者就可以借助内核构建系统来构建MTD内核驱动,并使读者指定的内核配置信息生效。

10.2 MTD 基础知识

现在我们已经在内核中进行了一个简单的MTD配置,接下来就可以检查这个子系统在Linux开发平台上是如何工作的。使用上一节中配置好的RAM测试驱动程序,我们就可以使用一个MTD设备来挂载JFFS2映像。假如读者已经按照第9章中所描述的方法创造了一个JFFS2映像,或许就想挂载并测试它。Linux内核并不支持ext2或其他文件系统那样将JFFS2文件系统映像直接挂载到回环设备上,所以必须另辟蹊径,即通过在支持MTD功能的Linux开发平台上使用MTD RAM测试驱动程序来实现这一点。如图10-1所示,代码清单10-3解释了相应步骤。

代码清单10-3 将JFFS2挂载到MTD内存设备上

```
# modprobe jffs2
# modprobe MTDblock
# modprobe MTDram
# dd if=jffs2.bin of=/dev/MTDblock0
4690+1 records in
```

```

4690+1 records out
# mkdir /mnt/flash
# mount -t jffs2 /dev/MTDblock0 /mnt/flash
# ls -l /mnt/flash
total 0
drwxr-xr-x  2 root root 0 Sep 17 22:02 bin
drwxr-xr-x  2 root root 0 Sep 17 21:59 dev
drwxr-xr-x  7 root root 0 Sep 17 15:31 etc
drwxr-xr-x  2 root root 0 Sep 17 15:31 home
drwxr-xr-x  2 root root 0 Sep 17 22:02 lib
drwxr-xr-x  2 root root 0 Sep 17 15:31 proc
drws----- 2 root root 0 Sep 17 15:31 root
drwxr-xr-x  2 root root 0 Sep 17 22:02 sbin
drwxrwxrwt  2 root root 0 Sep 17 15:31 tmp
drwxr-xr-x  9 root root 0 Sep 17 15:31 usr
drwxr-xr-x 14 root root 0 Sep 17 15:31 var
#

```

如代码清单10-3所示，首先安装Linux内核支持JFFS2和MTD子系统所需的可加载模块。在加载JFFS2模块之后还加载了mtddblock和mtdram模块。在加载系统必需的设备驱动模块之后，使用Linux的dd命令将JFFS2文件系统映像复制到MTD RAM中，然后通过使用mtdblock设备来测试驱动程序。本质上讲，我们使用了系统内存作为支持设备来模拟一个MTD块设备。

在将JFFS2文件系统映像复制到MTD块设备之后，就可以使用mount命令来挂载该JFFS2文件系统，其挂载方式如代码清单10-3所示。当MTD“虚拟设备”挂载之后，便可以随意操作这个JFFS2文件系统映像。使用这种方法的唯一限制是文件系统映像的大小不能改变。文件系统映像的大小受两个因素所限制：第一，在配置MTD RAM测试设备时，我们设定了它的最大容量为8MB^①；第二，当我们用mkfs.jffs2实用程序创建JFFS2映像时，也确定了映像的大小。当我们创建文件系统时，在文件夹中所指定的内容也就决定了文件系统映像的大小。可以参考前面第9章代码清单9-9中的内容，回想一下jffs2.bin映像是如何创建的。

我们必须清醒地认识到，采用这种方法检查JFFS2文件系统的种种限制，这一点非常重要。考虑一下我们所做过的工作：将一个文件（JFFS2文件系统二进制映像）中的内容复制到一个内核块设备（/dev/MTDblock0）中，然后将内核块设备（/dev/MTDblock）挂载为JFFS2文件系统。做完这些工作以后，就可以使用所有传统的文件系统操作命令来检查甚至修改文件系统。文件系统操作命令如ls、df、dh、mv、rm和cp等都可以用来检查和修改文件系统。但是，与回环设备不同的是，我们所复制的文件与已挂载的JFFS2文件系统映像之间并没有联系，因此，如果我们做出了一些修改之后再卸载文件系统，那么所做的这些修改内容将会丢失。如果想保留这些修改信息，就必须将其复制到一个文件里，其方法如下所示：

```
# dd if=/dev/MTDblock0 of=./your-modified-fs-image.bin
```

上述命令会生成一个名为your-modified-fs-image.bin的文件，其大小为我们在配置过程中指定的MTDblock0设备大小。在本例中，其大小为8MB。由于缺乏合适的JFFS2编辑工具，所

① 当我们在Linux内核配置中启用MTD RAM测试设备时，映像的大小已由内核配置确定。

以这是一个非常有效的检查和修改JFFS2文件系统的方法。更为重要的是，它说明了不带闪存设备的开发系统上MTD子系统的基础知识。现在让我们来看一些包含闪存物理设备的硬件。

配置 MTD

要将开发板上的闪存作为MTD设备来使用，就必须正确配置MTD。下面的清单列出了将板子上的闪存及相应设备正确配置为MTD系统所需做的工作。

- ☐ 指定闪存设备的分区；
- ☐ 指定闪存设备的类型和位置；
- ☐ 为选定的芯片配置合适的闪存驱动；
- ☐ 为内核配置合适的驱动。

接下来的章节将逐一探讨上述步骤。

10.3 MTD 分区

给定硬件平台上的大多数闪存设备都被分成几个部分，我们称之为分区，这与典型桌面工作站上硬盘的分区类似。MTD子系统提供对这种闪存分区的支持，但是MTD子系统必须被配置成支持MTD分区。图10-2举例说明了MTD分区支持的配置选项。

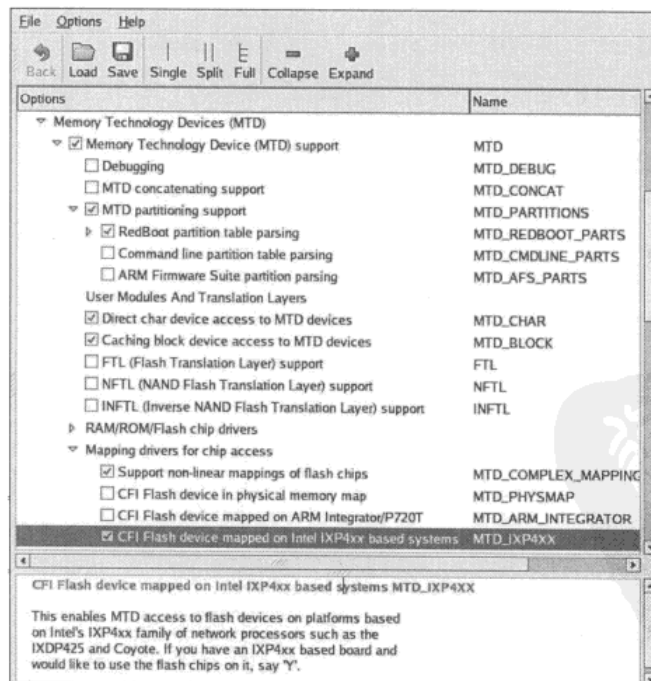


图10-2 MTD分区支持的内核配置

将分区数据传递给Linux内核的方法有好几种，目前支持的方法如下所示，在图10-2中的MTD

Partitioning Support选项下可以看到如下选项:

- ☐ Redboot partition table parsing;
- ☐ Kernel command-line partition table definition;
- ☐ Board-specific mapping drivers.

MTD也允许无分区数据的配置,在这种情况下,MTD会直接将整个闪存视为一个设备。

10.3.1 Redboot 分区表

定义和检测MTD分区的最为常用的方法来自于一个早先的实现方法:Redboot分区。Redboot是很多嵌入式开发板上所使用的一种引导装入程序,特别是ARM Xscale开发板,如ADI的Engineering Coyote Reference 开发平台。

MTD子系统定义了一种将分区信息存储到闪存设备本身的方法,这与硬盘分区表的概念很像。在Redboot分区的情况下,开发者保留并指定了一块闪存擦写块用来保存分区定义。当引导装入程序检测闪存设备的分区信息时,它将选用一个映射驱动程序来调用分区信息解析函数。图10-2显示了我们开发板的映射驱动程序,它最后被定义为CONFIG_MTD_IXP4xx。

通常来讲,仔细查看某个例子可帮助我们理解这些概念。我们首先要查看Coyote开发平台上的Redboot引导装入程序提供给我们信息。代码清单10-4的内容显示了一些Redboot引导装入程序在系统加电启动时的打印信息。

代码清单10-4 加电时的Redboot启动信息

```
Platform: ADI Coyote (XScale)
IDE/Parallel Port CPLD Version: 1.0
Copyright (C) 2000, 2001, 2002, Red Hat, Inc.

RAM: 0x00000000-0x04000000, 0x0001f960-0x03fd1000 available
Flash: 0x50000000 - 0x51000000, 128 blocks of 0x00020000 bytes each.
...
```

上述打印信息告诉我们,开发板上的RAM映射的物理起始地址为0x00000000,闪存映射的物理地址为0x50000000-0x51000000。我们还能看到闪存分为128个块,每个块的块大小为128KB。

Redboot包含一条命令用来创建和显示闪存上的分区信息。代码清单10-5的内容显示了fis list命令的输出结果,此命令为Redboot引导装入程序中闪存映像系统命令集的一部分。

代码清单10-5 Redboot 闪存分区列表

```
RedBoot> fis list
Name           Flash addr  Mem addr    Length     Entry point
RedBoot        0x50000000  0x50000000  0x00060000  0x00000000
RedBoot config 0x50FC0000  0x50FC0000  0x00001000  0x00000000
FIS directory  0x50FE0000  0x50FE0000  0x00020000  0x00000000
RedBoot>
```

从代码清单10-5中可以看到,Coyote开发板上定义了3个闪存分区。名为RedBoot的分区内包含了可执行的Redboot引导装入程序映像文件,名为RedBoot config的分区则包含了引导装入程

序所维护的配置参数，最后名为FIS directory的分区保存了闪存分区表信息。

进行适当的配置以后，Linux内核能够检测并解析这个分区表，同时创建MTD分区，该MTD分区表示了闪存存储设备的物理分区。代码清单10-6的内容列出了上述ADI Engineering Coyote开发板在启动时所打印的部分信息，所启动的Linux内核配置为支持检测Redboot分区功能。

代码清单10-6 Linux启动时检测到的Redboot分区

```
...
IXP4XX-Flash0: Found 1 x16 devices at 0x0 in 16-bit bank
Intel/Sharp Extended Query Table at 0x0031
Using buffer write method
cfi_cmdset_0001: Erase suspend on write enabled
Searching for RedBoot partition table in IXP4XX-Flash0 at offset 0xfe0000
3 RedBoot partitions found on MTD device IXP4XX-Flash0
Creating 3 MTD partitions on "IXP4XX-Flash0":
0x00000000-0x00060000: "RedBoot"
0x00fc0000-0x00fc1000: "RedBoot config"
0x00fe0000-0x01000000: "FIS directory"
...
```

当内核检测到闪存芯片以后，系统内核将打印出如代码清单10-6中所示的第一条信息，这是通过通用闪存接口（CFI，Common Flash Interface）驱动程序也即CONFIG_MTD_CFI来实现的。CFI是一个检测闪存芯片特征的行业标准，可以检测如制造商、设备类型、闪存总容量和擦写块容量这些特征信息。本章末尾的“参考资源”中会列出介绍CFI规范的网址。

启用CFI选项是通过内核配置工具中MTD选项下的顶层配置菜单来实现的，选择RAM/ROM/Flash chip drivers菜单下的Detect flash chips by Common Flash Interface (CFI) probe选项即启用CFI功能，配置过程如图10-3所示。

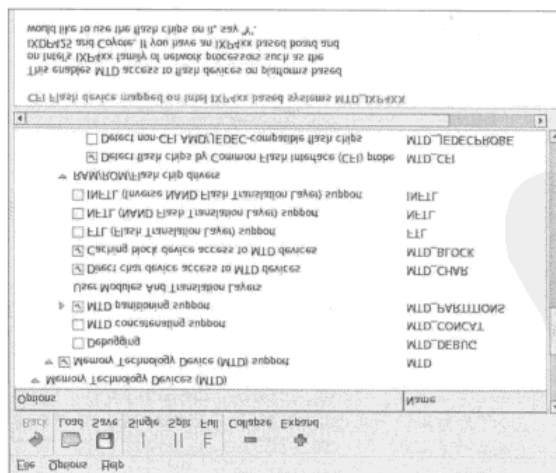


图10-3 MTD CFI支持的内核配置

如代码清单10-6所示，通过CFI接口就可以检测到闪存芯片。因为我们也启用了CONFIG_MTD_

REDBOOT_PARTS (参见图10-2), MTD将搜索闪存芯片中的Redboot分区表信息。可以注意到该芯片的设备名已经被枚举为IXP4XX-Flash0。读者可以从代码清单10-6看到Linux内核已经从闪存芯片中检测到三个分区,这与同前面使用的Redboot中使用fis list命令列出的信息一致。

在打印出分区组织信息的同时, Linux内核会自动检测并创建用于表示3个闪存分区的内核数据结构。当内核完成初始化以后,在/proc文件系统中将会看到这些分区信息,如代码清单10-7所示。

代码清单10-7 内核MTD 闪存分区信息

```
root@coyote:~# cat /proc/mtd
dev:      size  erasesize  name
mtd0: 00060000 00020000 "RedBoot"
mtd1: 00001000 00020000 "RedBoot config"
mtd2: 00020000 00020000 "FIS directory"
#
```

我们很容易就能创建一个新的Redboot分区。本例程中使用了Redboot的FIS命令,但是在此书中我们并没有详细解释Redboot命令的使用方法,读者如果感兴趣可以参考本章结尾“参考资源”中列出的Redboot用户参考文档。代码清单10-8的内容显示了创建一个新Redboot分区的细节。

代码清单10-8 创建一个新的Redboot分区

```
RedBoot> load -r -v -b 0x01008000 coyote-40-zImage
Using default protocol (TFTP)
Raw file loaded 0x01008000-0x0114dccb, assumed entry at 0x01008000
RedBoot> fis create -b 0x01008000 -l 0x145cd0 -f 0x50100000 MyKernel
... Erase from 0x50100000-0x50260000: .....
... Program from 0x01008000-0x0114dcd0 at 0x50100000: .....
... Unlock from 0x50fe0000-0x51000000: .
... Erase from 0x50fe0000-0x51000000: .
... Program from 0x03fd0000-0x03fff000 at 0x50fe0000: .
... Lock from 0x50fe0000-0x51000000: .
```

首先,我们需要加载用来创建新分区的映像文件。在本例中使用了我们的内核映像文件,将其加载到内存的0x0100800地址处。之后使用Redboot的fis create命令创建了新分区。我们已经使用Redboot命令在闪存区域的地址空间0x50100000处创建了新分区,读者可以看到Redboot将首先擦除这块闪存区域,并将内核映像文件写入此区域。在最后的操作次序中,Redboot将它的目录区域解锁,并采用新的分区信息来更新FIS目录。代码清单10-9的内容显示了使用fis list命令对新分区信息的输出结果,可以与代码清单10-5的结果做一比较。

代码清单10-9 新的Redboot 分区列表

```
RedBoot> fis list
Name          FLASH addr  Mem addr    Length     Entry point
RedBoot       0x50000000  0x50000000  0x00060000  0x00000000
RedBoot config 0x50FC0000  0x50FC0000  0x00001000  0x00000000
FIS directory  0x50FE0000  0x50FE0000  0x00020000  0x00000000
MyKernel      0x50100000  0x50100000  0x00160000  0x01008000
```

当然，引导Linux内核后，Linux内核将发现新的分区，并且此时我们可以对其做一些适当的操作。聪明的读者此时已经发现使用这个新分区的好处，那就是可以通过闪存来启动内核，而不需要每次都通过tftp的方式来加载内核。接下来将介绍这个从闪存启动的命令，使用Redboot的exec命令，并把闪存分区的起始地址和内核映像文件长度作为参数传递给它，就可以将内核文件复制到RAM。

```
RedBoot> exec -b 0x50100000 -l 0x145cd0
Uncompressing Linux..... done, booting the kernel.
...
```

10.3.2 内核命令行分区

如10.3节所述，原始闪存分区信息可以使用其他方法传递给内核。事实上，手动将分区信息直接写到Linux内核命令行也许不是最简单的方法，但也可能是最直接的方法。当然，就像我们刚刚了解到的，一些引导装入程序（例如U-Boot）使得分区过程较为简单。然而另外一些引导装入程序并不能在启动过程中将内核命令行参数传递给内核。在这种情况下，内核命令行参数必须在编译的时候就配置好，因此，命令行参数很难改变，只有在重新编译内核的时候，分区信息才能改变。

为了启用MTD子系统的命令行分区功能，必须对内核进行配置以支持这一功能。在图10-2中的MTD partitioning support选项下可以看到这个配置选项。请选择command-line partition table parsing选项，它对CONFIG_MTD_CMDLINE_PARTS配置选项进行了定义。

代码清单10-10的内容显示了在内核命令行中定义分区的格式。（取自.../drivers/MTD/cmdlinepart.c代码中的内容。）

代码清单10-10 内核命令行MTD分区格式

```
mtddparts=<mtdddef>[;<mtdddef>
* <mtdddef> := <mtid-id>:<partdef>[,<partdef>]
* <partdef> := <size>[@offset][<name>][ro]
* <mtid-id> := unique name used in mapping driver/device (mtd->name)
* <size> := std linux memsize OR "-" to denote all remaining space
* <name> := '(' NAME ')'
```

每一个传递给内核命令行的MTDdef参数都定义了一个独立的分区。如代码清单10-10所示，其中每一个mtdddef定义都包含了多个部分的内容，读者可以指定独一无二的ID、分区大小以及闪存起始地址的偏移量。也能够传递指定的分区名，并且还有一个可选择的只读特性。读者可以请重新参考代码清单10-5中的Redboot分区定义，在内核命令行中静态地定义这些参数，如下所示：

```
mtddparts=MainFlash:384K(Redboot),4K(config),128K(FIS),-(unused)
```

通过这些定义，使该内核具有了4个MTD分区，其MTD的ID为MainFlash，该定义也包含了闪存分区的大小和分布情况，这与代码清单10-5所示情况相符。

10.3.3 映射驱动程序

如果用户想要定义特定开发板上的闪存布局，最后一种方法是使用一个与该开发板相对应的映射驱动程序。在Linux的内核源码树中包含了很多映射驱动程序的示例，这些示例保存在.../drivers/MTD/maps目录下。这其中的每一个例子都能帮助读者创造自己的映射驱动程序，其执行细节随体系结构的不同而不同。

映射驱动程序是一个适当的内核模块，由module_init()和module_exit()这两个调用组成，如第8章的相关内容所述。一个典型的映射驱动程序很小而且很容易编译通过，只包含不到20几行的C代码。

代码清单10-11复制了.../drivers/MTD/maps/pq2fads驱动程序文件的一部分内容。这个映射驱动程序定义了Freescale PQ2FADS评估板上的闪存设备，同样也支持MPC8272和其他处理器。

代码清单10-11 PQ2FADS闪存映射驱动程序

```
...
static struct mtd_partition pq2fads_partitions[] = {
    {
#ifdef CONFIG_ADS8272
        .name      = "HRCW",
        .size      = 0x40000,
        .offset    = 0,
        .mask_flags = MTD_WRITEABLE, /* force read-only */
    }, {
        .name      = "User FS",
        .size      = 0x5c0000,
        .offset    = 0x40000,
#else
        .name      = "User FS",
        .size      = 0x600000,
        .offset    = 0,
#endif
    }, {
        .name      = "uImage",
        .size      = 0x100000,
        .offset    = 0x600000,
        .mask_flags = MTD_WRITEABLE, /* force read-only */
    }, {
        .name      = "bootloader",
        .size      = 0x40000,
        .offset    = 0x700000,
        .mask_flags = MTD_WRITEABLE, /* force read-only */
    }, {
        .name      = "bootloader env",
        .size      = 0x40000,
        .offset    = 0x740000,
        .mask_flags = MTD_WRITEABLE, /* force read-only */
    }
};
```

```

/* pointer to MPC885ADS board info data */
extern unsigned char __res[];

static int __init init_pq2fads_mtd(void)
{
    bd_t *bd = (bd_t *)__res;
    physmap_configure(bd->bi_flashstart, bd->bi_flashsize,
                     PQ2FADS_BANK_WIDTH, NULL);

    physmap_set_partitions(pq2fads_partitions,
                          sizeof (pq2fads_partitions) /
                          sizeof (pq2fads_partitions[0]));

    return 0;
}

static void __exit cleanup_pq2fads_mtd(void)
{
}

module_init(init_pq2fads_mtd);
module_exit(cleanup_pq2fads_mtd);
...

```

这个示例虽很简单,但是Linux设备驱动程序将完整地把PQ2FADS闪存映射传递到MTD子系统。可以回顾第8章的相关内容,当设备驱动程序中的一个函数使用宏`module_init()`声明以后,在Linux内核启动的某个适当时刻这个函数将自动被调用。在PQ2FADS映射驱动程序中,模块的初始化函数`init_pq2fads_MTD()`仅仅进行了如下两次简单的调用。

- `physmap_configure()` 会将闪存芯片的物理地址、大小、每个数据bank的宽度信息以及访问闪存设备所需的任何设置函数都传递给MTD子系统。
- `physmap_set_partitions()` 将开发板的分区信息传递给MTD子系统,分区信息来自`pq2fads_partitions[]`数组中定义的分表,这个数组可在映射驱动程序的起始处找到。

通过学习这些简单例程,你能为自己的开发板设计映射驱动程序。

10.3.4 闪存芯片驱动程序

目前MTD已能支持大量不同种类的闪存芯片和设备。如果够幸运,也许你选用的芯片也在支持之列。大部分常见的闪存芯片都支持前面提到的CFI。较老的闪存芯片可能会支持JEDEC,这是一个比较早的闪存兼容标准。图10-4显示了取自最新Linux内核快照的内核配置。这个版本的Linux内核支持很多类型的闪存。

如果你的闪存芯片不在支持之列,那就必须自行提供设备文件。在`.../drivers/MTD/chips`目录下有很多例子,你可以从中挑选一个,来定制或创建自己的闪存设备驱动程序。幸运的是,如果你的闪存芯片不是那些较新的且带有最新接口的类型,很有可能已经有人写好了你所需的闪存驱动。

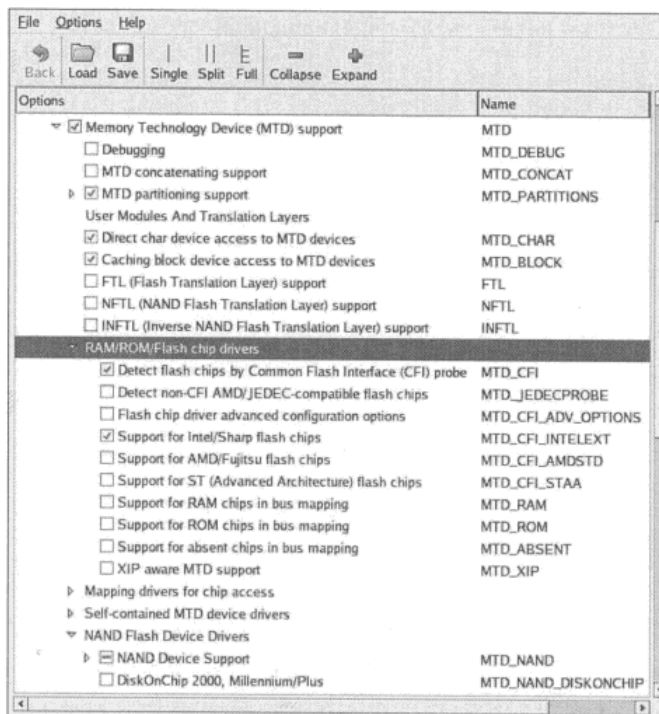


图10-4 闪存设备支持

10.3.5 特定开发板的初始化

10

除了映射驱动程序以外，特定开发板（平台）的设置程序必须提供MTD 闪存系统操作的基础定义。代码清单10-12的内容复制了.../arch/arm/mach-ixp4xx/coyote-setup.c文件中的相关部分。

代码清单10-12 Coyote特定开发板上的设置

```
static struct flash_platform_data coyote_flash_data = {
    .map_name = "cfi_probe",
    .width    = 2,
};

static struct resource coyote_flash_resource = {
    .start    = COYOTE_Flash_BASE,
    .end      = COYOTE_Flash_BASE + COYOTE_Flash_SIZE - 1,
    .flags    = IORESOURCE_MEM,
};

static struct platform_device coyote_flash = {
    .name      = "IXP4XX-Flash",
    .id        = 0,
    .dev       = {
```

```

        .platform_data = &coyote_flash_data,
    },
    .num_resources      = 1,
    .resource           = &coyote_flash_resource,
};

...

static struct platform_device *coyote_devices[] __initdata = {
    &coyote_flash,
    &coyote_uart
};

static void __init coyote_init(void)
{
    ...

    platform_add_devices(coyote_devices,
                        ARRAY_SIZE(coyote_devices));
}
...

```

代码清单10-12仅给出了coyote-setup.c文件中平台初始化文件的部分相关内容。从该代码清单的末尾处开始看，coyote_init()函数调用了platform_add_devices()函数，这个函数指定了在文件前面提到的Coyote开发板特定的设备定义。在coyote_init()函数前面，你会发现文件定义了两个设备。其中我们感兴趣的是coyote_flash。这个platform_device结构包含了Linux内核和MTD子系统所需的所有重要细节信息。

coyote_flash数据结构的.name成员将特定平台的闪存资源与具有相同名字的映射驱动程序绑定起来。读者可以在映射驱动程序文件../drivers/MTD/maps/ixp4xx.c中看到它。该数据结构的.resource成员则传递开发板的闪存基地址。数据结构的.dev成员还包含了.platform_data成员，将闪存设置同芯片驱动程序绑定在一起。通过这些方法，我们已经通过将内核配置为CONFIG_MTD_CFI来指定开发板使用CFI检测方法检测闪存。读者可以在图10-4中查看相关配置选择。

读者可以使用与此相近的方法，根据自己的体系结构和开发板来为其定义闪存支持。

10.4 MTD 实用程序

MTD包包含了一系列有用的系统实用程序，用来配置和管理MTD子系统。MTD子系统应该在用户的Linux内核源码树内构建，而这些实用程序则应与MTD子系统分开构建。这些实用程序可以使用一种与交叉编译其他任何用户应用程序相类似的方法来构建。

在使用这些实用程序时必须特别注意，因为它们并没有错误保护机制。实用程序的一点点错误就可能对用户硬件开发平台的引导装入程序被全部擦除，除非已经进行了备份，并且知道如何使用JTAG闪存编程工具重新烧写引导装入程序，否则这将毁掉用户一整天的工作。

为了将实践贯穿本书，我们不可能用太多篇幅来介绍每一个MTD实用程序，因此只是着重

介绍其中最为常用和有用的实用程序，并将其他实用程序的使用留作练习。最新的MTD快照包含了超过20个二进制应用程序。

实用程序中的flash_*系列程序对于原始的MTD设备分区非常有用。该系列包括flashcp、flash_erase、flash_info、flash_lock、flash_unlock等，各个实用程序的作用不言自明。当这些分区被定义并且被作为内核设备列出以后，用户空间下的任何应用程序就可以在某一个分区上运行。再次重申：如果在包含引导装入程序的分区上运行flash_erase命令，那么将得到一片空白区。如果你希望做这样的实验，最好先备份引导装入程序映像文件，并知道如何使用JTAG硬件仿真器或其他JTAG编程工具重新烧写引导装入程序。

在Coyote开发板上所运行的内核中看一下代码清单10-8（Mykernel）中所创建的新分区，分区信息如代码清单10-13所述。在这里可以看到在内核设备MTD1上新建立的分区。

代码清单10-13 内核MTD 分区列表

```
root@coyote:~# cat /proc/mtd
dev:      size  erasesize  name
mtd0: 00060000 00020000 "RedBoot"
mtd1: 00160000 00020000 "MyKernel"
mtd2: 00001000 00020000 "RedBoot config"
mtd3: 00020000 00020000 "FIS directory"
```

使用MTD实用程序，可以在新建立的分区上执行一系列的操作。下面介绍在分区上使用flash_erase实用程序的执行结果。

```
# flash_erase /dev/mtd1
Erase Total 1 Units
Performing Flash Erase of length 131072 at offset 0x0 done
```

如果想将一个新的内核映像文件复制到这个分区，可以使用flashcp命令：

```
root@coyote:~# flashcp /workspace/coyote-40-zImage /dev/mtd1
```

在根文件系统分区上工作要有意思一些。我们已经可以选择用引导装入程序或者Linux内核将初始化映像放置在Redboot闪存分区上。首先，我们用Redboot创建一个新分区来保存根文件系统。下面的命令会在闪存中建立一个名为RootFS的新分区，其起始物理地址为0x50300000，该分区具有30个存储块。需要记住的是，一个存储块即为闪存的一个擦除单元，具体到这个闪存芯片上为128K。

```
RedBoot> fis create -f 0x50300000 -l 0x600000 -n RootFS
```

接下来，对内核进行引导并将根文件系统映像文件复制到名为RootFS的新分区上。在读者目标板的Linux命令提示行下输入下列命令可完成此功能。需要注意这样做的前提是假设已经将文件系统映像放置在开发板上的可访问目录下。正如本书多次提到的，NFS是用于开发的最好选择。

```
root@coyote:~# flashcp /rootfs.ext2 /dev/mtd2
```

文件系统可以放在任何位置，大小从2MB到所分配分区的最大容量，所以这个命令的执行需

要一些时间。记住的是，该操作包括将映像文件烧写到闪存。复制结束以后可以将此分区挂载为一个文件系统。代码清单10-14的内容显示了具体操作顺序。

代码清单10-14 将MTD 闪存分区挂载为ext2文件系统

```
root@coyote:~# mount -t ext2/dev/mtdblock2 /mnt/remote ro
root@coyote:~# ls -l /mnt/remote/
total 16
drwxr-xr-x 2 root root 1024 Nov 19 2005 bin
drwxr-xr-x 2 root root 1024 Oct 26 2005 boot
drwxr-xr-x 2 root root 1024 Nov 19 2005 dev
drwxr-xr-x 5 root root 1024 Nov 19 2005 etc
drwxr-xr-x 2 root root 1024 Oct 26 2005 home
drwxr-xr-x 3 root root 1024 Nov 19 2005 lib
drwxr-xr-x 3 root root 1024 Nov 19 2005 mnt
drwxr-xr-x 2 root root 1024 Oct 26 2005 opt
drwxr-xr-x 2 root root 1024 Oct 26 2005 proc
drwxr-xr-x 2 root root 1024 Oct 26 2005 root
drwxr-xr-x 2 root root 1024 Nov 19 2005/sbin
drwxr-xr-x 2 root root 1024 Oct 26 2005/srv
drwxr-xr-x 2 root root 1024 Oct 26 2005/sys
drwxr-xr-x 2 root root 1024 Oct 26 2005/tmp
drwxr-xr-x 6 root root 1024 Oct 26 2005/usr
drwxr-xr-x 2 root root 1024 Nov 19 2005/var
root@coyote:~#
```

代码清单10-14中有两个巧妙之处。注意，我们在文件系统挂载命令中所指定的设备类型为/dev/MTDblock2。MTD块驱动程序使我们使用块设备的方式访问MTD分区。用/dev/MTD2作为指定设备将告诉内核使用MTD字符设备驱动程序。mtdchar和mtdblock都是虚拟的设备驱动程序，用来提供字符方式或者块方式访问底层的闪存分区。因为要挂载块设备，就必须将参数指定为块设备。图10-1显示了启用这些访问方式的Linux内核配置。CONFIG_MTD_CHAR和CONFIG_MTD_BLOCK为相应的Linux内核配置宏定义。

第二个巧妙之处在于使用只读(ro)参数设置mount命令。为了实现只读而采用MTD块模拟驱动程序从闪存里挂载ext2文件系统映像，是可以接受的。但是，系统并不支持用mtdblock驱动程序对ext2设备进行写操作，因为ext2并不能够识别Flash擦除块。为了对一个基于闪存的文件系统进行写访问，需要使用一个能够识别闪存的文件系统，如JFFS2。

JFFS2 根文件系统

建立JFFS2根文件系统是一个简单明了的过程。除了压缩机制以外，JFFS2还支持磨损平衡机制。磨损平衡机制的设计初衷是通过将擦写周期均匀地分配到设备存储块，以增加闪存设备的使用周期。就如第9章所指出的，闪存的寿命受有限擦写次数的影响。磨损平衡可以看作是所使用的任何基于闪存文件系统必须具有的特征。正如本书某些地方所探讨的一样，可以将闪存看成是一个不是经常进行写操作的物理介质。特别需要指出的是，读者应该避免使用那些需要对目标板的闪存文件系统进行频繁写操作的程序。要特别留意那些日志程序，如syslogd。

我们可以在开发平台上使用Redboot RootFS分区中所用到的ext2文件系统映像，来建立JFFS2文件系统映像。JFFS2文件系统的压缩优点将立刻显露出来。在以前的那个RootFS例程中，我们使用的是ext2文件系统映像。下面是此文件的详细信息：

```
# ls -l rootfs.ext2
-rw-r--r-- 1 root root 6291456 Nov 19 16:21 rootfs.ext2
```

下面使用MTD包内的mkfs.jffs2实用程序将这个文件系统映像转换为JFFS2格式。代码清单10-15中列出了命令和结果。

代码清单10-15 将RootFS 转换为JFFS2

```
# mount -o loop rootfs.ext2/mnt/flash/
# mkfs.jffs2 -r /mnt/flash -e 128 -b -o rootfs.jffs2
# ls -l rootfs.jffs2
-rw-r--r-- 1 root root 2401512 Nov 20 10:08 rootfs.jffs2
#
```

首先将ext2文件系统映像挂载到一个回环设备上，这个回环设备位于开发平台上的一个专用挂载点。然后调用mkfs.jffs2实用程序来创建JFFS2文件系统映像文件。其中的-r选项指示mkfs.jffs2实用程序来确认根文件系统映像的存放位置。-e选项指示mkfs.jffs2实用程序假定以每个存储块大小为128KB来创建映像文件，默认大小为64KB。如果用户的闪存设备包含一个和映像文件大小不同的存储块，那么JFFS2就不能发挥它的最大效率。最后，我们列出一个较长的列表，并发现最终所得到的JFFS2根文件系统映像的大小已经缩减了60%。当使用了一个存储容量有限的闪存时，就会发现这在珍贵的闪存资源的占用上可以大量缩减。

请注意在代码清单10-15中那个传递给mkfs.jffs2的重要命令行标志。-b代表大端模式（-big-endian）标志，这表示要使用mkfs.jffs2实用程序在一个具有大端模式的目标板上创建一个JFFS2 闪存映像。因为我们的目标板是ADI Engineering Coyote开发板，这个开发板上的Intel IXP425处理器运行于大端模式，所以这一步对于以后正确的操作至关重要。如果没有指定大端模式，那么在当内核尝试访问JFFS2文件系统超级块时，你将会看到内核会在屏幕上打印出很多乱码^①。有人会猜出为什么我会记得这些细节？

使用和上一个例子相似的方式，我们可以使用flashcp实用程序将这个映像文件复制到Redboot RootFS Flash分区。然后可以使用JFFS2根文件系统来启动Linux内核。代码清单10-16显示了在我们的目标板硬件上运行MTD实用程序的细节。

代码清单10-16 将JFFS2复制到RootFS分区

```
root@coyote:~# cat /proc/mtd
dev:      size  erasesize  name
mtd0: 00060000 00020000 "RedBoot"
mtd1: 00160000 00020000 "MyKernel"
mtd2: 00600000 00020000 "RootFS"
```

① 内核可以配置为一个错误的大小端MTD文件系统，但是这会降低系统性能。在一些配置中（如多处理器结构），这将会是一个有用的特性。

```

mtd3: 00001000 00020000 "RedBoot config"
mtd4: 00020000 00020000 "FIS directory"
root@coyote:~# flash_erase /dev/mtd2
Erase Total 1 Units
Performing Flash Erase of length 131072 at offset 0x0 done
root@coyote:~# flashcp /rootfs.jffs2 /dev/mtd2
root@coyote:~#

```

这里，在内核配置中启用JFFS2文件系统非常重要，即运行`make ARCH=<arch> gconfig`，并在其配置工具中的File Systems菜单下选择JFFS2选项。另一个有用的提示是在MTD实用程序中使用`-v` (verbose) 标志位，这将在对闪存进行操作时提供一些进程更新及其他有用信息。

我们刚刚已经看到如何使用Redboot的`exec`命令来引导内核。代码清单10-17的内容详细显示了使用新的JFFS2文件系统作为根文件系统来加载和引导Linux内核后的命令信息。

代码清单10-17 用JFFS2作为根文件系统来启动Linux

```

RedBoot> load -r -v -b 0x01008000 coyote-zImage
Using default protocol (TFTP)
Raw file loaded 0x01008000-0x0114decb, assumed entry at 0x01008000
RedBoot> exec -c "console=ttyS0,115200 rootfstype=jffs2 root=/dev/mtdblock2"
Using base address 0x01008000 and length 0x00145ecc
Uncompressing Linux..... done, booting the kernel.
...

```

10.5 小结

- MTD子系统为在Linux内核中使用存储器设备（如闪存）提供了必要支持。
- 使用MTD之前必须在Linux内核配置中启用MTD功能支持，本章几幅插图详细列举了各配置选项。
- 作为MTD内核配置的一部分，读者必须为自己选用的闪存芯片选择合适的驱动程序。图10-4显示了在一个最新Linux内核快照所支持的闪存驱动程序集。
- 闪存存储设备可以作为一个大型设备来进行管理，也可以划分成多个分区使用。
- 可以运用多种方法来将闪存分区信息传递给Linux内核，包括使用Redboot分区信息、内核命令行参数以及映射驱动程序等方法。
- 用户特定开发板提供了映射驱动程序及其定义，向内核确定所用闪存的配置。
- MTD 和一系列用户空间实用程序用来管理闪存设备上的映像文件。
- 日志闪存文件系统（JFFS2）是一个小型、高效的基于闪存的文件系统，可以视为MTD子系统的助手工具，在本章中，我们创建了一个JFFS2映像，并在目标板上将其挂载为根文件系统。

参考资源

MTD Linux主页
www.linux-mtd.infradead.org/

Redboot用户手册

<http://ecos.sourceware.org/ecos/docs-latest/redboot/redboot-guide.html>

通用闪存接口规范

AMD公司

www.amd.com/us-en/assets/content_type/DownloadableAssets/cfi_r20.pdf



本章内容

- BusyBox简介
- BusyBox配置
- BusyBox操作
- 小结

BusyBox的主页将BusyBox喻为“嵌入式Linux里的瑞士军刀”。这个比喻非常恰当，相对于庞大的标准Linux命令行实用程序而言，BusyBox不仅小巧而且高效，往往可作为构建资源受限的嵌入式平台的基础。本章将介绍BusyBox，为我们定制BusyBox安装打下良好的基础。

此前我们已多次略带提及BusyBox，本章将详细介绍这个非常有用的软件包。在简要介绍BusyBox后，接着会探讨BusyBox的配置工具，该工具用于裁减BusyBox以满足特定的需求；之后将讨论交叉编译BusyBox软件包所需的条件。

本章还介绍了BusyBox的一些操作细节，包括在嵌入式系统里的具体用法。此外，我们将检查BusyBox的初始化过程，并且解释它与标准System V初始化过程的不同之处。这一节还会给出一个BusyBox初始化脚本示例。在学习将BusyBox安装到目标系统的详细步骤之后，读者将了解BusyBox的若干命令及其局限性。

11.1 BusyBox 简介

BusyBox在嵌入式Linux社区中赢得了巨大的声誉。无论是配置、编译还是使用都非常容易，同时还能显著减少为支持大量常用Linux实用程序所需的总的系统资源。BusyBox提供了一些紧凑的程序，可以替代那些大多数桌面和嵌入式Linux发行版中使用的成熟实用程序。例如，ls、cat、cp、dir、head和tail等文件工具；dmesg、kill、halt、fdisk、mount、umount等常用实用程序……不一而足。BusyBox还支持一些更为复杂的操作，如ifconfig、netstat、route及其他网络工具。

BusyBox具有模块化和高度可配置的特点，可以进行裁减以满足项目的特定需求。BusyBox提供的配置实用程序与配置Linux内核所用的实用程序类似，因此BusyBox的配置很容易上手。

与其对应的全功能版本相比，BusyBox里命令的实现一般更为简单。在某些情况下，BusyBox

仅支持一部分常用的命令行选项。不过，实际上你会发现BusyBox提供的命令功能子集足以满足绝大多数嵌入式系统的常见需求。

BusyBox 容易使用

如果读者熟悉Linux内核的配置和编译，就会发现BusyBox的配置、编译和安装也非常简单直观，这些步骤非常相似：

- (1) 执行配置程序并选取需要的特性；
- (2) 运行make dep命令构建依赖关系树；
- (3) 运行make命令编译软件包；
- (4) 将二进制程序和一系列符号链接^①安装到目标系统上。

你可以在开发用工作站或嵌入式目标系统上构建并安装BusyBox软件包。BusyBox在这两种环境中都能很好地工作。不过，在开发工作站上安装时需要多加小心，最好将BusyBox隔离在一个单独的工作目录中，以免覆盖系统原有的启动文件或重要工具。

11.2 BusyBox 配置

要进行BusyBox配置，可键入下面的命令，与使用Linux内核基于ncurses库的配置实用程序相同：

```
$ make menuconfig
```

图11-1展示了BusyBox的顶层配置菜单。

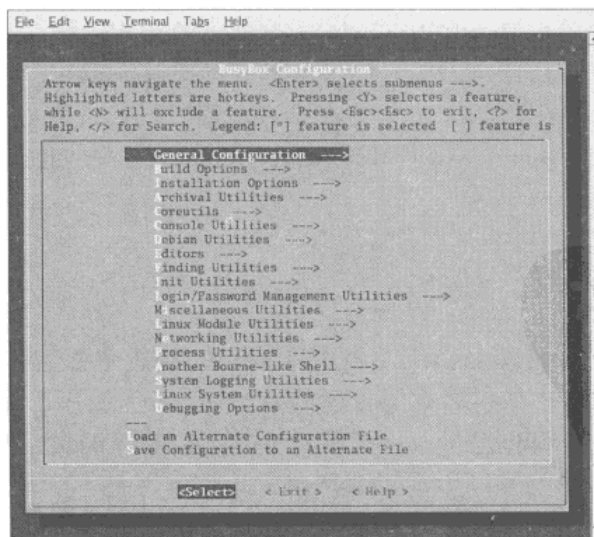


图11-1 BusyBox顶层配置菜单

^① 稍候我们会详细介绍符号链接。

限于篇幅，无法一一列出所有配置选项，不过其中有些配置选项确实值得一提。在Build Options菜单里，可以找到若干重要的BusyBox配置选项，包括交叉编译BusyBox必需的配置选项。代码清单11-1详细列出了最新版BusyBox（快照）的Build Options下的编译选项。在BusyBox的顶层菜单中选择Build Options项即可进入该配置界面。

代码清单11-1 BusyBox的Build Options编译选项

```
[ ] Build BusyBox as a static binary (no shared libs)
[ ] Build with Large File Support (for accessing files > 2 GB)
[ ] Do you want to build BusyBox with a Cross Compiler?
() Any extra CFLAGS options for the compiler?
```

第一个选项用于构建极小型嵌入式系统，该选项支持以静态方式编译和链接BusyBox，这样目标系统上就不再需要动态加载库（例如libc-2.3.3.so）。如果不选择该选项，BusyBox需要一些共享库的支持才能运行。使用ldd命令，很容易就能确定目标系统的BusyBox（或其他二进制程序）要依赖哪些库的支持。代码清单11-2给出了在桌面Linux工作站上显示的输出。

代码清单11-2 BusyBox的依赖库

```
$ ldd busybox
linux-gate.so.1 => (0xfffffe000)
libc.so.6=> /lib/tls/libc.so.6 (0x42c70000)
/lib/ld-linux.so.2=> /lib/ld-linux.so.2 (0x42c57000)
```

注意上面的BusyBox工具利用默认的配置选项进行编译，因此需要代码清单11-2所示的3个共享库的支持。如果选择以静态方式构建BusyBox，执行ldd命令则直接给出诸如“BusyBox不是一个动态可执行程序”的提示信息。换句话说，它在执行过程中不需要共享库的支持（即通过共享库解析任何未决依赖）。由于不需要共享库的支持，静态链接的二进制在根文件系统中只产生较小的影响。不过，如果要构建一个不使用共享库的嵌入式应用程序，也就意味着在应用程序里无法使用那些熟悉的C库函数。

下一节将介绍代码清单11-1列出的其他选项。

交叉编译 BusyBox

正如本章开头提到的，BusyBox的作者有意将BusyBox用在交叉开发环境中，因此在这样的环境中编译BusyBox非常容易。在大多数情况下，仅需指定开发平台中交叉编译器的前缀(prefix)。这需要在BusyBox配置工具的Build Options中选择使用交叉编译器编译一项，然后在给出的选项中输入交叉编译器的前缀。输入的前缀取决于使用的交叉开发环境，例如xscale_be-或ppc_Linux-。我们将在下一章介绍嵌入式开发环境时深入介绍相关内容。

代码清单11-1中的最后一个选项用来向编译器命令行指定一些额外的标记，可能包括生成调试信息(-g)、设置优化级别（例如-O2）和其他一些可能针对特定安装或目标系统所需的编译选项。

11.3 BusyBox 操作

在构建好BusyBox之后，最终会得到一个名为busybox的二进制程序。可以通过这个二进制程序名本身调用BusyBox，不过更常见的方式是通过其符号链接（symlink）进行调用。BusyBox程序执行时不带命令行参数，就会输出一个配置时选定的功能列表，其执行结果如代码清单11-3所示（为满足页面宽度要求已做过一定调整）。

代码清单11-3 BusyBox用法

```
root@coyote # ./busybox
BusyBox v1.01 (2005.12.03-18:00+0000) multi-call binary

Usage: busybox [function] [arguments]...
      Or: [function] [arguments]...

BusyBox is a multi-call binary that combines many common Unix
Utilities into a single executable. Most people will create a
link to busybox for each function they wish to use and BusyBox
Will act like whatever it was invoked as!

Currently defined functions:
[, ash, basename, bunzip2, busybox, bzip, cat, chgrp, chmod,
chown, chroot, chvt, clear, cmp, cp, cut, date, dd, deallocvt, df, dirname, dmesg,
du, echo, egrep, env, expr, false, fgrep,
find, free, grep, gunzip, gzip, halt, head, hexdump, hostname,
id, ifconfig, init, install, kill, killall, klogd, Linuxrc, ln,
logger, ls, mkdir, mknod, mktemp, more, mount, mv, openvt, pidof,
ping, pivot_root, poweroff, ps, pwd, readlink, reboot, reset,
rm, rmdir, route, sed, sh, sleep, sort, strings, swapoff, swapon,
sync, syslogd, tail, tar, tee, test, time, touch, tr, true, tty,
umount, uname, uniq, unzip, uptime, usleep, vi, wc, wget, which,
whoami, xargs, yes, zcat
```

通过代码清单11-3，可以看到这次BusyBox构建中启用的功能列表。这些功能按照字母顺序从ash（一个专为减少内存消耗而优化过的shell）到zcat（一种用于解压缩压缩文件内容的实用程序）依次列出，是这个最新版BusyBox（快照）默认启用的实用程序集。

要调用BusyBox某个特定功能，只需在（命令行中）执行busybox时紧跟一个已定义的命令。例如要显示当前目录下的文件清单，可以执行如下命令：

```
[root@coyote]# ./busybox ls
```

从代码清单11-3的BusyBox使用信息中得到的另一个重要内容是对该程序的简要描述。它把BusyBox描述为一个可重复调用的二进制程序，即将大量常用实用程序组合成一个可执行程序；这也是先前提到的符号链接之目的所在。BusyBox可以被一个以所要执行程序功能命名的符号链接所调用，这样在调用一个指定功能时就不必键入两个单词的命令，此外它还给使用者提供了一系列命名类似的实用程序集，代码清单11-4和代码清单11-5清晰地表明了这一点。

代码清单11-4 BusyBox符号链接顶层结构

```
[root@coyote]$ ls -l /
total 12
drwxrwxr-x  2 root  root 4096 Dec  3 13:38 bin
lrwxrwxrwx  1 root  root  11 Dec  3 13:38 Linuxrc -> bin/busybox
drwxrwxr-x  2 root  root 4096 Dec  3 13:38 sbin
drwxrwxr-x  4 root  root 4096 Dec  3 13:38 usr
```

代码清单11-4展示了BusyBox软件包使用make install命令所建立的目标目录结构。可执行文件busybox位于/bin目录下，该路径其余目录都会包含有指向/bin/busybox的符号链接，代码清单11-5扩展了代码清单11-4的目录结构。

代码清单11-5 BusyBox符号链接树结构详细内容

```
[root@coyote]$ tree
.
|-- bin
|   |-- ash -> busybox
|   |-- busybox
|   |-- cat -> busybox
|   |-- cp -> busybox
|   |-- ...
|   |-- zcat -> busybox
|-- Linuxrc -> bin/busybox
|-- sbin
|   |-- halt -> ../bin/busybox
|   |-- ifconfig -> ../bin/busybox
|   |-- init -> ../bin/busybox
|   |-- klogd -> ../bin/busybox
|   |-- ...
|   |-- syslogd -> ../bin/busybox
'-- usr
    |-- bin
    |   |-- [ -> ../../bin/busybox
    |   |-- basename -> ../../bin/busybox
    |   |-- ...
    |   |-- xargs -> ../../bin/busybox
    |   |-- yes -> ../../bin/busybox
    |-- sbin
    |   |-- chroot -> ../../bin/busybox
```

为了提高可读性，避免占用长达3页的篇幅，代码清单11-5中的输出已做了大幅缩减。包含一个省略号(...)的每一行表示该部分已经被省略，只给出了给定路径下最初部分和最后部分的目录内容。实际上，在这些目录下添加了100多个符号链接，实际数量取决于用户在使用BusyBox配置实用程序时所启用的功能选项。

注意可执行程序busybox本身，它处在/bin目录下第二项。此外，/bin目录下还包含指向busybox的符号链接，例如ash、cat、cp、...、zcat等。为增加可读性，cp和zcat之间的项已被略去。通过这个符号链接结构表，用户可以直接输入那些实用程序名来调用相应的功能。例如，要使用busybox ifconfig实用程序配置一个网络接口，可以输入这样的命令：

```
$ ifconfig eth1 192.168.1.14
```

该实用程序将通过ifconfig符号链接调用可执行程序busybox。BusyBox会检查自身(busybox)是如何被调用的,即读取argv[0]的内容以决定执行什么功能。

11.3.1 BusyBox之init

注意代码清单11-5中名为init的符号链接,第6章已讲解了init程序及其在系统初始化中的作用。我们可以回想起在内核初始化的最后阶段内核试图执行一个称为/sbin/init的程序。毫无疑问,BusyBox也可以模仿这个初始化(init)功能,就像代码清单11-5描述的系统,BusyBox可以很好地实现初始化功能。

BusyBox处理系统初始化的过程有别于标准的System V初始化。在第6章中描述的一个使用System V (SysV)初始化的Linux系统,需要在/etc目录下有一个inittab文件。BusyBox也会读取一个inittab文件,但是两个inittab文件的语法是有区别的。一般情况下,当用户使用BusyBox时并不需要使用inittab文件。我同意BusyBox帮助手册对此的观点:如果需要运行级,最好使用System V初始化^①。

让我们来看看在嵌入式系统中是怎样的。我们已经构建了一个基于BusyBox的小根文件系统,并把BusyBox配置为静态链接方式,不需要任何共享库。我们采用9.10节描述的步骤构建了这个小文件系统,这里不再详细介绍这一过程。代码清单11-6列出了这个简单文件系统里的文件。

代码清单11-6 最小的BusyBox根文件系统

```
$ tree
.
|-- bin
|   |-- busybox
|   |-- cat -> busybox
|   |-- dmesg -> busybox
|   |-- echo -> busybox
|   |-- hostname -> busybox
|   |-- ls -> busybox
|   |-- ps -> busybox
|   |-- pwd -> busybox
|   '-- sh -> busybox
|-- dev
|-- '--- console
|-- etc
|-- proc
4 directories, 10 files
```

这个基于BusyBox的根文件系统占据的空间和busybox程序本身大小差别不大。使用这种静态链接方式配置的BusyBox支持大约100种实用程序,可执行的BusyBox要小于1MB:

```
# ls -l /bin/busybox
-rwxr-xr-x 1 root root 824724 Dec 3 2005 /bin/busybox
```

^① 第6章详细介绍了System V初始化。

现在来看看这个系统是如何运转的。代码清单11-7显示了这个基于BusyBox的嵌入式系统在加电后控制台的输出信息。

代码清单11-7 BusyBox默认启动过程

```
...
Looking up port of RPC 100003/2 on 192.168.1.9
Looking up port of RPC 100005/1 on 192.168.1.9
VFS: Mounted root (nfs filesystem).
Freeing init memory: 96K
Bummer, could not run '/etc/init.d/rcS': No such file or directory

Please press Enter to activate this console.

BusyBox v1.01 (2005.12.03-19:09+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

-sh: can't access tty; job control turned off
/ #
```

代码清单11-7中的例子运行在一个用NFS挂载方式配置的嵌入式开发板上。在工作站上输出了一个目录，该目录包含了代码清单11-6中详细描述简单根文件系统映像文件。在引导过程的最后阶段之一，目标板上的Linux内核通过NFS方式挂载一个根文件系统。当内核试图执行/sbin/init的时候，它失败了，这是因为在文件系统中没有/sbin/init程序。然而，就像代码清单11-7中可以看到，内核也会试图执行/bin/sh。这在用BusyBox配置的目标系统中成功执行了，通过根文件系统中的符号链接/bin/sh，busybox得以运行。

BusyBox显示的第一件事情是报告它没有找到/etc/init.d/rcS脚本文件，这是BusyBox查找的默认初始化脚本文件。不使用inittab文件，这是基于BusyBox的嵌入式系统首选的初始化方法。

当初始化过程完成后，BusyBox给出一个提示使用户按回车键来激活控制台。当检测到有“回车”键按下后，就会执行ash shell来等待用户的输入信息。BusyBox显示的最后一条信息是关于作业控制的，这是在一个串行终端设备上创建系统控制台的结果。Linux内核包含这样的代码，即如果检测到控制台设置为在一个串行终端上运行就会禁止作业控制。

上面的例子可以生成一个可运行的系统，该系统会提供大约100个可用的Linux实用程序，包括核心实用程序、文件实用程序、网络支持实用程序和一个具有相当功能的shell。你可以看到，这个简单的软件包提供了一个强大的平台来构建用户自己的系统应用程序。当然，也需要说明，不会有任何libc库和其他库函数的支持，所以用户在实现自己的应用程序时将面临艰巨的任务，因为用户将不得不提供对所有常用的系统调用和一个典型C程序所依赖的一些库函数的支持。可供选择的是，用户可以静态链接用户应用程序所依赖的库，但是如果在用户的应用程序较多的情况下采用这种静态链接方法，用户的应用程序将可能会超过用户目标系统上动态链接库和用户目标系统上共享库组合后的大小。

11.3.2 rcs 初始化脚本示例

在BusyBox产生一个交互的shell之前，会执行名为/etc/init.d/rcs脚本文件中的命令，如代码清单11-7所示。在一个基于BusyBox系统中，用户的应用程序就是在这个脚本文件中得以实现。代码清单11-8提供了一个简单的rcs初始化脚本文件。

代码清单11-8 简单的rcs BusyBox启动脚本

```
#!/bin/sh

echo "Mounting proc"
mount -t proc /proc /proc

echo "Starting system loggers"
syslogd
klogd

echo "Configuring loopback interface"
ifconfig lo 127.0.0.1

echo "Starting inetd"
xinetd

# start a shell
busybox sh
```

这个简单的脚本不需要做太多说明。首先，在其自身保留的/proc挂载点处挂载/proc文件系统非常重要，这是因为在/proc文件系统下可以得到许多实用程序的相关信息，这在第9章中有更加详细的解释。接下来，为了发现系统的启动问题，我们可以尽可能早地启动系统日志。启动运行后台的核心守护进程之后，我们可以为系统配置本地的回环接口。许多传统的Linux实用程序都假定有回环接口，如果用户系统支持socket配置，那么这个假定的接口就是可用的。在启动shell之前要做的最后一件事情，是启动系统因特网超级守护进程（xinetd），该程序会在系统后台监听任何经过配置的网络接口设备的网络请求。例如，为了开启与开发板的telnet会话，xinetd会接收telnet连接请求并且生成一个telnet服务来处理该会话。

用户的应用程序可以通过启动初始化脚本rcs而不是启动一个shell来执行。代码清单11-8是一个具有Telnet服务目标板的简单示例，在该目标板上运行着一些基本的服务，如系统和内核的日志管理。

11.3.3 在目标平台安装 BusyBox

对于BusyBox安装的讨论，只有当读者理解了符号链接的用法和目的后，才可以继续下去。在BusyBox的makefile文件中含有一个称为install的目标文件，执行make install命令后会创建一个包含可执行busybox程序和一个符号链接树的目录结构。该环境连同链接树都需要移植到用户嵌入式系统的根目录下。采用链接树后就没有必要在每个命令中输入busybox。例如，要在

给定目录下查看文件代码清单，用户就可以仅仅输入ls命令。ls的符号链接会执行前面所述的busybox程序并且调用ls功能。回顾代码清单11-4和代码清单11-5中的链接树，可以注意到采用BusyBox构建的系统仅会创建那些在BusyBox配置程序中所选择功能选项的符号链接。

要创建一个带有必要符号链接的根文件系统，最简单的方法是让BusyBox为用户构建根文件系统，这样用户仅需简单地将根文件系统挂载到开发平台上，并且将一个PREFIX参数传递到BusyBox的makefile文件中。代码清单11-9给出了这一过程。

代码清单11-9 BusyBox安装到根文件系统

```
$ mount -o loop bbrootfs.ext2 /mnt/remote
$ make PREFIX=/mnt/remote install
/bin/sh applets/install.sh /mnt/remote
/mnt/remote/bin/ash -> busybox
/mnt/remote/bin/cat -> busybox
/mnt/remote/bin/chgrp -> busybox
/mnt/remote/bin/chmod -> busybox
/mnt/remote/bin/chown -> busybox
...
/mnt/remote/usr/bin/xargs -> ../../bin/busybox
/mnt/remote/usr/bin/yes -> ../../bin/busybox
/mnt/remote/usr/sbin/chroot -> ../../bin/busybox

-----
You will probably need to make your busybox binary
setuid root to ensure all configured applets will
work properly.
-----

$ chmod +s /mnt/remote/bin/busybox
$ ls -l /mnt/remote/bin/busybox
-rwsr-sr-x 1 root root 863188 Dec 4 15:54 /mnt/remote/bin/busybox
-----
```

首先把根文件的二进制映像文件挂载到期望的挂载点处，例如本例中我所偏好的/mnt/remote目录下，然后调用BusyBox的make install命令来传递PREFIX参数，这样可以指定符号链接树和可执行busybox程序的存放地点。就像代码清单11-9所示，makefile通过调用一个称作applets/install.sh的脚本来做这些大量的工作。该脚本会创建一个包含所有可用BusyBox应用程序的文件，同时会在使用PREFIX参数指定的路径下，为每一个程序创建相应的符号链接。这个脚本非常繁琐，每一个创建的符号链接都会占用脚本的一行输出。为了简便起见，代码清单11-9中只显示了开始和最后声明的几个符号链接，中间的省略号代表省略的内容。

在安装脚本中也显示了关于setuid的信息，这是为了提醒用户，也许有必要为busybox设置setuid，以便超级用户之外的用户也具有相关权限。当然这并不是绝对必需的，特别是在一个嵌入式Linux的环境中，因为在嵌入式系统中只有一个根用户存在的情形非常普遍。但是，如果必须这样做（非根用户也可以调用busybox程序），那么需要使用代码清单11-9中的命令（chmod+s）来实现。

这个安装过程的结果就是在我们的目标根文件系统上安装了busybox二进制程序和其符号链接树，该结果和代码清单11-4中所示内容非常相似。

在这里有必要指出，BusyBox也有在运行时可以在目标系统上创建符号链接树的功能，该功能可以在配置BusyBox时选中，并且通过执行带有-install参数的busybox程序时调用。不过用户想实现这样的用法，必须把/proc文件系统挂载到用户的目标系统。

11.3.4 BusyBox 命令

在最新的BusyBox版本中，其帮助手册记录了197条命令（也称作applet）。这些命令足够用来支持相当复杂的shell脚本了，包括bash shell。BusyBox已经支持了awk和sed功能，你可以在Bash脚本中经常看到它们。BusyBox也支持诸如ping、ifconfig、traceroute和netstat这样的网络实用程序。另外一些命令，包括true、false和yes也可以明确支持shell脚本。

可以花一些时间仔细阅读附录B中的BusyBox命令，那里简单介绍了每个BusyBox命令。看过这些内容之后，你将会对BusyBox的功能以及BusyBox如何在自己的嵌入式Linux项目中使用有更好的理解。

就像本章开始提到的，与全功能的传统Linux实用程序相比，许多BusyBox命令的功能具有一定的局限性。通常情况下，通过在调用命令时加入--help选项，你就可以在执行BusyBox命令过程中得到任意给定BusyBox命令的帮助信息。该操作可以输出每一种BusyBox命令使用方法的简单描述信息。BusyBox的gzip applet就是一个很好的例子，它有助于理解BusyBox命令的局限性。代码清单11-10显示了一个基于BusyBox平台下执行gzip -help命令后的输出内容。

代码清单11-10 BusyBox的gzip用法

```
/ # gzip --help
BusyBox v1.01 (2005.12.01-21:11+0000) multi-call binary

Usage: gzip [OPTION]... [FILE]...

Compress FILE(s) with maximum compression.
When FILE is '-' or unspecified, reads standard input. Implies -c.

Options:
  -c      Write output to standard output instead of FILE.gz
  -d      Decompress
  -f      Force write when destination is a terminal
```

BusyBox的gzip命令只支持三个命令行选项，对应的全功能版本则支持超过15个的命令行参数。例如，全功能版本的gzip命令支持--list选项，使用该选项可以在命令行中产生压缩包中的文件代码清单。BusyBox中的gzip命令则不支持该选项。当然，对于嵌入式系统而言，这通常算不上重大缺陷。之所以提及这一点，旨在帮助读者在面对BusyBox时作出明智的选择。如果需要使用某个实用程序的全部功能，解决办法也很简单：在配置BusyBox时去掉对该实用程序的支持，并在目标系统里添加对应的标准Linux实用程序。

11.4 小结

- BusyBox是一个用于嵌入式系统的强大工具,通过一个、具有多种调用方式的二进制程序,可以替代大量常用的Linux实用程序;
- BusyBox能显著减小根文件系统映像的大小;
- BusyBox易于使用并且具有大量非常有用的特性;
- BusyBox的配置工具使用了与Linux内核配置类似的界面,简单直观;
- 根据用户的特定需求, BusyBox可配置成静态或动态链接的应用程序;
- 利用BusyBox实现的系统初始化与常见的有一定区别,本章介绍了这些区别;
- BusyBox提供了大量命令。附录B详细列出了一个最新版BusyBox支持的全部命令。

参考资源

BusyBox 项目主页

www.busybox.net/

BusyBox帮助手册

www.busybox.net/downloads/BusyBox.html



本章内容

- 交叉开发环境
- 主机系统需求
- 为目标板提供服务
- 小结

作为一名嵌入式开发人员，主机开发系统上启用的配置和服务对项目的成功具有极大影响。本章将探讨交叉开发环境特有的需求，以及嵌入式开发人员为提高效率而需要了解的一些工具和技巧。

一开始探讨一个典型的交叉开发环境。通过熟知的“Hello World”实例，详细说明基于主机的应用程序与针对嵌入式系统的应用程序之间的显著差异；同时分析了本机（native）和嵌入式应用程序开发所用工具链之间的不同之处。接着提出主机系统要求的必备条件，详细介绍了主机系统中一些重要组成部分的使用方法。最后，介绍了一个目标板示例，该目标板由基于网络的主机提供服务。

12.1 交叉开发环境

刚刚接触嵌入式开发的开发人员通常会苦苦思索本机和交叉开发环境之间的概念和区别。确实，通常会有3种编译器和3个（甚至更多）版本的标准头文件，如stdlib.h。如果缺少合适的工具和主机端实用程序，在目标嵌入式系统上调试应用程序会很困难。你必须设法管理设计用于运行在主机系统上的文件和实用程序，并将它们与用于目标板的文件和实用程序区分开。

我们在这里使用的主机（host）一词指的是开发工作站，也即运行着某个Linux桌面发行版的桌面电脑。相反，这里使用的目标（target）一词指的是嵌入式硬件平台。因此，本地开发（native development）指的是在主机系统上为其自身编译并构建应用的过程，交叉开发指的是在主机系统上编译并构建将在嵌入式系统上运行的应用的过程。牢记这些定义有助于你正确理解本章内容。

图12-1显示了典型的交叉开发环境的布局。PC主机通过一种或多种物理连接方式与目标板相连接，如果目标板的串口和以太网接口都可用就再方便不过了。在后面讨论内核调试时，你会发现第2个串口将非常有用。

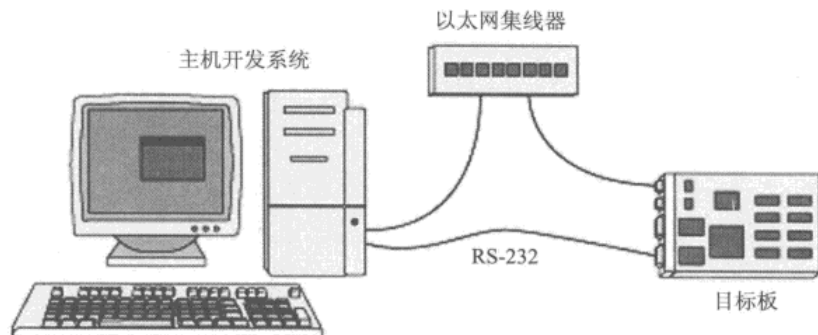


图12-1 交叉开发设置

最常见的情形是，开发人员在主机上会开一个串口终端连到RS-232串口上，也许是通过一个或多个Telnet终端会话连接到目标板，也许是一个或多个通过以太网作为连接介质的调试会话。这种交叉开发设置提供了极大的灵活性。基本思路是主机系统提供运行编译器、调试器、编辑器和其他实用程序的环境，而目标板仅执行专为其设计的应用。没错，你当然可以在目标系统上运行编译器和调试器，但是我们假定你的主机提供更多的资源，包括RAM、磁盘存储和因特网连接。实际上，对于嵌入式目标板来说，缺少人机输入设备或输出显示设备并不罕见。

“Hello World” 嵌入式程序

一套配置合理的交叉开发系统会对应用开发人员隐藏大量错综复杂的内容，下面这个简单的实例将为我们揭示并解释其中一些奥秘。当编译简单的“Hello World”程序时，工具链（编译器、链接器和相关实用程序）会对我们正用来创建程序的主机系统和准备编译的程序做大量假设。实际上也并不是假设，而是编译器创建二进制文件的一系列规则。

代码清单12-1给出了一个简单的“Hello World”程序。

代码清单12-1 Hello World

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}
```

毫无疑问，即使再不专业的应用开发人员也能了解这段C代码中的一些要点。首先，调用printf()函数，它并未在该文件中定义。如果略去包含printf()函数原型的#include指令，编译器会给出熟悉的消息：

```
hello.c:5: warning: implicit declaration of function 'printf'
```

这就带来一些有趣的问题：

□ `stdio.h` 文件在哪里？我们如何找到它？

□ `printf()` 函数在哪里运行？它在二进制可执行文件中是如何解析的？

看起来编译器只知道以某种方式创建一个在命令行上可执行的二进制文件，更复杂的是，最终的可执行文件会包含我们从未见过的启动和终止序言（*prologue*）代码，这些代码由链接器自动包含到可执行文件。这些代码负责处理如下细节，包括传递给程序的环境和参数、启动和终止的一些常规事务，退出处理，等等。

欲编译“Hello World”程序，只需调用一个简单的编译器命令，如下所示：

```
$ gcc -o hello hello.c
```

上述命令会生成一个名为`hello`的二进制可执行文件，可以直接从命令行执行。编译器使用的默认设置会告诉编译器去哪里查找头文件。与之类似，对`printf()`函数引用的解析也很简单，链接器只要包含定义有该函数的库的引用即可。当然，这里是标准C库。

我们可以查询工具链以查看一些用到的默认设置。代码清单12-2列出了使用`-v`选项后的`cpp`命令的部分输出。你也许已经知道`cpp`是`gcc`工具链的C预处理器组件。为增加可读性，格式上做了部分调整（只涉及空格）。

代码清单12-2 本地`cpp`的默认搜索路径

```
$ cpp -v
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/3.3.3/specs
Configured with: ../configure --prefix=/usr
↳ --mandir=/usr/share/man --infodir=/usr/share/info
↳ --enable-shared --enable-threads=posix --disable-checking
↳ --disable-libunwind-exceptions --with-system-zlib
↳ --enable-__cxa_atexit-host=i386-redhat-linux

Thread model: posix
gcc version 3.3.3 20040412 (Red Hat Linux 3.3.3-7)
/usr/lib/gcc-lib/i386-redhat-linux/3.3.3/ccl -E -quiet -v -
ignoring nonexistent directory "/usr/i386-redhat-linux/include"

#include "...": search starts here:
#include <...>: search starts here:
  /usr/local/include
  /usr/lib/gcc-lib/i386-redhat-linux/3.3.3/include
  /usr/include
End of search list.
/usr/lib/
```

这个简单的查询给出了一些非常有用的信息。首先可以看到，编译器是如何使用我们熟悉的`./configure`实用程序进行配置的。其次，默认的线程模型是`posix`，如果你使用了线程函数，这项值决定应用程序所链接的线程库。最后，你会看到`#include`指令的默认搜索路径。

但是，如果打算为不同的体系结构（如PowerPC）编译`hello.c`，又该怎么办？在主机上使用交叉编译器为PowerPC目标板编译应用程序时，必须确定编译器不会使用默认的主机头文件

(include) 目录或库搜索路径。选用一个已正确配置的交叉编译器是第一步，设计完善的交叉开发环境是第二步。

代码清单12-3来自一个流行的开源交叉工具链的输出，即众所周知的嵌入式Linux开发工具包（Embedded Linux Development Kit, ELDK），它由Denx软件工程公司开发并维护。这个特殊版本专为PowerPC 82xx工具链配置。同样，为了便于阅读，输出里增加了一些空格。

代码清单12-3 默认交叉搜索路径

```
$ ppc_82xx-cpp -v
Reading specs from /opt/eldk/usr/bin/..
↳ /lib/gcc-lib/ppc-linux/3.3.3/specs

Configured with: ../configure --prefix=/usr
↳ --mandir=/usr/share/man --infodir=/usr/share/info
↳ --enable-shared--enable-threads=posix --disable-checking --with-system-zlib
↳ --enable-__cxa_atexit --with-newlib --enable-languages=c,c++ --disable-libgcj
↳ --host=i386-redhat-linux --target=ppc-linux

Thread model: posix

gcc version 3.3.3 (DENX ELDK 3.1.1 3.3.3-10)
/opt/eldk/usr/bin/.. /lib/gcc-lib/ppc-linux/3.3.3/cc1
↳ -E -quiet -v -iprefix /opt/eldk/usr/bin/..
↳ /lib/gcc-lib/ppc-linux/3.3.3/ -D__unix__ -Dgnu_linux__
↳ -D__linux__ -Dunix -D__unix -Dlinux -D__linux -Asystem=unix
↳ -Asystem=posix -mcpu=603

ignoring nonexistent directory "/opt/eldk/usr/ppc-linux/sys-include"
ignoring nonexistent directory "/opt/eldk/usr/ppc-linux/include"
#include "... search starts here:

#include <...> search starts here:
/opt/eldk/usr/lib/gcc-lib/ppc-linux/3.3.3/include
/opt/eldk/ppc_82xx/usr/include

End of search list.
```

这里可以看到，include目录的默认搜索路径已作调整，指向了交叉版本而不是本机的include目录。这个看似不起眼的细节对于开发嵌入式系统应用和编译开源软件包是至关重要的。对于刚接触嵌入式系统但具备非常丰富的开发经验的应用开发人员来说，这也是最容易混淆的主题之一。

12.2 主机系统需求

开发工作站必须包括几个重要的组件和系统。当然，你需要正确配置交叉工具链。你可以下载和自己编译，或者从众多可用的商用工具链中获得一份。自行构建工具链已经超出本书范畴，不过有几本很好的参考资料，参见本章最后的“参考资源”。

下一个必备的重要组件是针对你的嵌入式系统体系结构定制的Linux发行版，它包括成百上千甚至成千上万个构成你的嵌入式系统文件系统的文件。同样，可以自行创建或获取一个商业版本。在互联网上很流行的一个嵌入式系统发行版是前面提到的ELDK，它适用于一些PowerPC和其他嵌入式平台。讲述从零开始构建一个嵌入式Linux发行版这一话题就足够写一本书，而且这些内容也超出了本书讨论的范畴。

总之，开发主机需要四种截然不同的能力：

- 交叉工具链和库；
- 目标系统软件包，包括程序、实用程序和库；
- 主机工具，例如编辑器、调试器和实用程序；
- 为目标板提供的服务（将在下一节进行介绍）。

如果你在自己的工作站上安装了现成的嵌入式Linux开发环境，不论是商业版还是可从开源社区自动获取的版本，工具链和组件都已预先配置好以便能协同工作。例如，工具链已经配置了默认的搜索路径目录，该目录会与目标平台头文件和开发工作站的系统库相匹配。如果你的开发工作站需要包括支持多种体系结构和处理器，那情况将更为复杂。这就是嵌入式Linux发行版存在的原因。

硬件调试探针

除了前面列出的组件之外，你应该考虑一些基于硬件的调试类型，它由连接到你的主机（通常通过以太网）和通过目标板上的调试连接器连接到目标平台的硬件探针组成。现在市场上有许多解决方案，第14章将详细讨论。

12.3 为目标板提供服务

参考图12-1，你会注意到从嵌入式目标板到主机开发系统的以太网连接。尽管严格上讲这不是必需的，而且实际上有些较小的嵌入式设备根本不带以太网接口，但这只是个别现象而非普遍情形。在目标板上提供以太网连接，对得起你在芯片上的花费。

在开发内核的过程中，你将多次编译内核并将其下载到嵌入式开发板中。许多嵌入式开发系统和引导装入程序都支持TFTP协议，并假定开发人员会使用之。TFTP是一个轻量级协议，用于TFTP服务器和客户端之间文件传输，这一点和FTP类似。

在引导装入程序里使用TFTP下载内核比使用串口（哪怕是在较高波特率下）下载内核要节省很多等待时间。载入ramdisk将花费更长的时间，因为ramdisk映像根据需求可能大到数十兆字节或更大。用在配置和使用TFTP上的时间花费肯定有回报，因此强烈推荐使用这种方式。极少会有设计在开发过程中不提供以太网接口，即使在真正投产时要去掉以太网口。

12.3.1 TFTP 服务器

在Linux开发主机上配置TFTP服务并不难。当然，根据你为开发工作站选择的Linux发行版的不同，其配置细节也有差异。本书给出的方法基于Red Hat和Fedora Core Linux发行版。

TFTP是一种TCP/IP服务,因此工作站必须开启TCP/IP服务。要开启TFTP服务,必须指示服务器响应即将到来的TFTP数据包,并创建TFTP服务器。在许多Linux发行版中,可以编辑xinetd网络超级服务器使用的配置文件实现。例如,在Red Hat和Fedora桌面Linux发行版中,该文件是/etc/xinetd.d/tftp。代码清单12-4列出了从Fedora Core 2开发工作站提取的开启TFTP服务的配置,为排版需要做了一定的微调。

代码清单12-4 TFTP配置

```
# default: off
# description: The tftp server serves files using the trivial
# file transfer protocol. The tftp protocol is often used to
# boot diskless workstations, download configuration files to
# network-aware printers, and to start the installation process
# for some operating systems.

service tftp
{
    socket_type          = dgram
    protocol             = udp
    wait                = yes
    user                 = root
    server               = /usr/sbin/in.tftpd
    server_args          = -c -s /tftpboot
    disable              = no
    per_source           = 11
    cps                  = 100 2
    flags                = IPv4
}
```

在这个典型的设置中,TFTP服务已经开启(disable=no),并将服务文件配置到工作站的/tftpboot目录下。当xinetd网络超级服务器收到一个TFTP请求时,它会根据配置创建指定的服务(/usr/sbin/in.tftpd)。server_args指定的命令行参数传递到in.tftpd进程。在这种情况下,-s参数会告诉in.tftpd切换到指定的目录下(/tftpboot)。`-c`选项允许创建新文件,如果要从目标板向服务器写文件,这个选项就非常有用。

请参考随桌面Linux发行版提供的文档以了解与环境相关的细节。

12.3.2 BOOTP/DHCP 服务器

在开发主机里提供DHCP服务器可以简化嵌入式目标板的配置管理。我们已经达成这样一个共识:在目标硬件上提供以太网接口是一个好主意。当Linux从目标板启动时,以太网接口在使用前需要进行配置。此外,如果目标板使用NFS挂载根文件系统配置,Linux在启动过程完成前需要配置目标板的以太网接口。我们在第9章中介绍了NFS。

通常,Linux在启动过程中可以用两种方法初始化以太网/IP接口:

- 在Linux内核命令行或默认配置里直接指定固定的以太网接口参数。

□ 配置内核，以便启动时自动检测网络设置。

显而易见，后一个选择最为灵活。DHCP或BOOTP是目标板和服务器用来实现网络设置自动检测的协议，有关DHCP或BOOTP协议的细节请参阅本章末的“参考资源”。

DHCP服务器控制IP子网预先配置好的IP地址分配，DHCP或BOOTP客户端将根据配置加入进来。DHCP服务器监听来自DHCP客户端（例如目标开发板）的请求，为客户端分配地址和其他相关的信息，这也是它启动的一部分工作。在启动DHCP服务器时，可以使用-d调试选项检查一次典型的DHCP交换（如代码清单12-5）过程，并观察目标机请求配置的输出。

代码清单12-5 典型的DHCP交换

```
tgt> DHCPDISCOVER from 00:09:5b:65:1d:d5 via eth0
svr> DHCPOFFER on 192.168.0.9 to 00:09:5b:65:1d:d5 via eth0
tgt> DHCPREQUEST for 192.168.0.9 (192.168.0.1) from \
      00:09:5b:65:1d:d5 via eth0
svr> DHCPACK on 192.168.0.9 to 00:09:5b:65:1d:d5 via eth0
```

该序列开始是客户端（目标）发出的一个广播帧，试图找到DHCP服务器。如代码清单12-5中DHCPDISCOVER消息所示。服务器提供一个IP地址响应给客户（如果配置好并启用），从DHCPOFFER消息中可以明显看到。随后，客户端在本地测试该IP作为响应。测试过程包括向DHCP服务器发送DHCPREQUEST包，如上所示。最后，服务器确认这个分配给客户的IP地址，从而完成自动目标配置。

注意，正确配置的客户将会记住DHCP服务器分配的上次地址，这一点很有趣。下次启动时，假定客户机用的是服务器上次分配的IP地址，那么它会略过DHCPDISCOVER阶段，直接前进到DHCPREQUEST阶段。启动的Linux内核没有这项功能，启动时每次会执行相同的功能序列。

为主机配置DHCP服务器并不困难。通常，我们的建议是参考你使用的桌面Linux发行版附带的文档。在Red Hat或者Fedora Core发行版中，单个目标平台的配置项如代码清单12-6所示。

代码清单12-6 DHCP 服务器配置示例

```
# Example DHCP Server configuration
allow bootp;

subnet 192.168.1.0 netmask 255.255.255.0 {
    default-lease-time 1209600;      # two weeks
    option routers 192.168.1.1;
    option domain-name-servers 1.2.3.4;
    group {
        host pdna1 {
            hardware ethernet 00:30:bd:2a:26:1f;
            fixed-address 192.168.1.68;
            filename "uImage-pdna";
            option root-path "/home/chris/sandbox/pdna-target";
        }
    }
}
```

这是一个简单的示例，只是想说明能传给目标系统的信息。该信息包含目标板上网卡MAC地址与所分配的IP地址间的一一映射。除了它的静态IP地址外，还可以把其他信息传递给目标板。本例中，将默认的路由器和DNS服务器地址传递给了目标板，一起传递的还包含所选择文件的文件名，以及内核挂载的NFS根文件系统的根目录路径。这个文件名可能被引导装入程序使用，用来加载通过TFTP服务器获取的内核映像。也可以通过配置DHCP服务器来分配预定范围内的IP地址，但是使用如代码清单12-6中所示的静态地址会非常方便。

必须在Linux开发工作stations上开启DHCP服务器。通常可以使用主菜单或通过命令行的方式完成设置。参考使用的Linux发行版中提供的文档，以了解适合你的环境的细节。例如，在Fedora Core 2 Linux发行版中，开启DHCP服务的命令是在命令行提示符下简单地输入下面的命令：

```
$ /etc/init.d/dhcpd start (or restart)
```

除非配置了自动启动，否则每次都必须在工作站启动后执行这条命令。

安装一台DHCP服务器时，涉及许多细节。除非服务器位于一个私有的网络内，否则建议在使用之前先找系统管理员做检查。如果你和公司共用一个局域网，你将极有可能干扰公司的DHCP服务。

12.3.3 NFS 服务器

使用NFS挂载目标板根文件系统是非常有效的开发手段。开发过程中使用这个配置的优势如下：

- 根文件系统大小不受开发板上资源限制，例如闪存。
- 开发过程中，应用程序文件的修改将在目标系统中立即生效。
- 可以在开发和调试根文件系统之前调试和启动内核。

根据使用的桌面Linux发行版不同，架设一台NFS服务器也会有所区别。和本章介绍的其他服务一样，你必须参考所使用的Linux发行版中提供的文档以适合你的配置。NFS服务必须通过启动脚本、图形菜单或命令行的方式启动。例如Fedora Core 2 Linux桌面系统中，在根命令提示符下启动NFS服务的命令是：

```
$ /etc/init.d/nfs start (or restart)
```

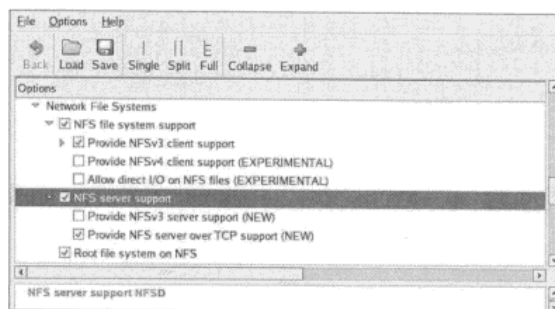


图12-2 NFS的内核配置

必须在每次启动Linux工作站时执行这个命令。（它和其他服务都可以在启动时自动运行，请参考桌面Linux发行版的文档。）另外，要启动该服务，必须将NFS编译进内核。DHCP和TFTP都是用户空间的实用程序，但是NFS需要内核的支持。无论是开发工作站还是目标板，内核都需要支持NFS。图12-2说明了内核中NFS的配置选项。注意，这些配置选项既支持NFS服务器，又支持客户端。还要注

意NFS根文件系统的选项。目标板内核必须配置为挂载NFS根文件系统。

NFS服务从服务器上的exports文件中获得它的指令，该文件一般为/etc/exports。代码清单12-7是一个简单的exports文件实例。

代码清单12-7 简单的NFS exports文件

```
$ cat /etc/exports
# /etc/exports
/home/chris/sandbox/coyote-target *(rw, sync, no_root_squash)
/home/chris/sandbox/pdna-target *(rw, sync, no_root_squash)
/home/chris/workspace *(rw, sync, no_root_squash)
```

在我的工作站上的这些项允许客户远程挂载代码清单12-7所示的三个目录中的任意一个。跟在路径后面的属性指示NFS服务器允许来自任何IP地址(*)的连接，并挂载各自具有给定属性的目录（使用no_root_squash读/写）。后一个属性允许具有根用户权限的客户在给定的目录中行使用那些权限。这在嵌入式系统的工作中通常是需要的，因为嵌入式系统通常只有根用户账户。

可以从工作站测试你的NFS配置是否正确。假定开启了NFS服务（需要NFS服务器和客户端组件），你可以挂载一个NFS文件系统，就如同挂载任何其他文件系统一样。

```
# mount -t nfs localhost:/home/chris/workspace /mnt/remote
```

如果这个命令成功执行，并且.../workspace中的文件出现在/mnt/remote，则说明你的NFS服务器配置已经工作了。

12.3.4 使用 NFS 为目标板挂载根文件系统

通过NFS挂载目标板的根文件系统并不难，正如之前提到的，这是一个非常有效的开发配置。不过在NFS能工作之前必须正确配置几个细节，所需步骤如下：

- (1) 根据所使用的体系结构配置NFS服务器，并导出正确的目标文件系统。
- (2) 配置目标内核支持NFS客户服务，配置“Root file system on NFS”选项。
- (3) 开启目标板以太网接口的“kernel-level autoconfiguration（内核级自动配置）”选项。
- (4) 通过内核命令行或内核配置选项配置以太网IP地址。
- (5) 提供内核命令行以开启NFS服务。

在解释NFS服务器配置时，我们用图12-2说明了内核配置。必须确保你的目标内核配置了NFS客户端服务，特别是要选中“Root file system on NFS”选项。确信内核选项中配置了CONFIG_NFS_FS=y和CONFIG_ROOT_NFS=y。显然，如果想要启动NFS挂载根文件系统，你就不能将NFS配置成可加载模块。

“kernel-level autoconfiguration”是一个TCP/IP配置选项，它在内核配置实用程序中的Networking选项卡下。当在目标内核中选中CONFIG_IP_PNP选项时，你会看到几个自动配置的选项，选择前面提到的BOOTP或DHCP。图12-3说明了“kernel-level autoconfiguration”内核配置。

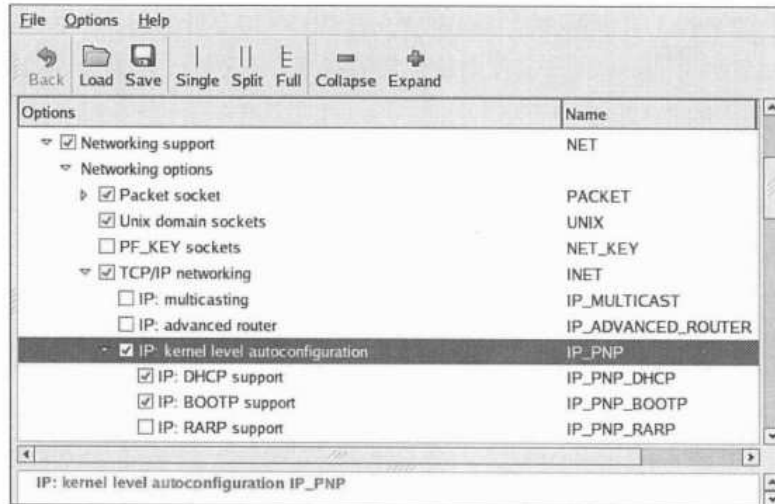


图12-3 kernel-level autoconfiguration选项

当配置好服务器和目标板内核后，需要使用前面介绍的一种方法对目标板的以太网进行配置。如果引导装入程序支持内核命令行，那么它是最简单的一种配置方法。下面是用于支持NFS根文件系统挂载的命令行：

```
console=ttyS0,115200 root=/dev/nfs rw ip=dhcp \
nfsroot=192.168.1.9:/home/chris/sandbox/pdna-target
```

12.3.5 U-Boot NFS 根挂载示例

U-Boot是一个很好的引导装入程序示例，它支持可配置的内核命令行。利用U-Boot的非易失环境特性，我们可以把命令行以一个参数的形式保存起来。要在U-Boot中使用NFS命令行，执行下面的命令（全部内容在串行端口终端的同一行上）：

```
setenv bootargs console=ttyS0,115200 root=/dev/nfs rw \
ip=dhcp nfsroot=192.168.1.9:/home/chris/sandbox/pdna-target
```

然后通过TFTP服务器加载内核。代码清单12-8列出了在一个嵌入式PowerPC目标板上的输出。

代码清单12-8 通过TFTP服务器加载内核

```
=> tftpboot 200000 uImage-pdna <<< Entered at U-Boot prompt
Using FEC ETHERNET device
TFTP from server 192.168.1.9; our IP address is 192.168.1.68
Filename 'uImage-pdna'.
Load address: 0x200000
Loading: #####
#####
#####
done
```

```
Bytes transferred = 911984 (dea70 hex)
=>
```

当启动内核时，可以看到详细的NFS根文件系统配置。代码清单12-9复制了所选择的内核启动消息的输出，以此来说明这个配置。为增加可读性，该输出做了格式处理（省略了许多行并增加了一些空格）。

代码清单12-9 使用NFS挂载的启动方式

```
Uncompressing Kernel Image ... OK
Linux version 2.6.14 (chris@pluto) (gcc version 3.3.3
↳ (DENX ELDK 3.1.1 3.3.3-10)) #1 Mon Jan 2 11:58:48 EST 2006
.
.
Kernel command line: console=ttyS0,115200 root=/dev/nfs rw
↳ nfsroot=192.168.1.9:/home /chris/sandbox/pdna-target ip=dhcp
.
.
Sending DHCP requests ... OK
IP-Config: Got DHCP answer from 192.168.1.9, my address is 192.168.1.68
IP-Config: Complete:
    device=eth0, addr=192.168.1.68, mask=255.255.255.0,
    gw=255.255.255.255, host=192.168.1.68, domain=,
    nis-domain=(none), bootserver=192.168.1.9,
    rootserver=192.168.1.9,
    rootpath=/home/chris/sandbox/pdna-target
.
.
Looking up port of RPC 100003/2 on 192.168.1.9
Looking up port of RPC 100005/1 on 192.168.1.9
VFS: Mounted root (nfs filesystem).
.
.

BusyBox v0.60.5 (2005.06.07-07:03+0000) Built-in shell (msh)
Enter 'help' for a list of built-in commands.

#
```

从代码清单12-9中，首先可以看到内核命令行后的标语。这里详细说明这个内核命令行中的4个选项：

- ☐ 控制台设备 (/dev/console);
- ☐ 根设备 (/dev/nfs);
- ☐ NFS根路径 (/home/chris/sandbox/pdna-target);
- ☐ “IP kernel-level autoconfiguration” 方法 (dhcp)。

然后，我们看到内核通过DHCP尝试内核级自动配置。当完成服务器响应以及DHCP交换时，内核显示下面命令行中检测到的配置。你可以从该代码中看到DHCP服务器已经分配了目标板的IP地址192.168.1.68。将检测到的配置与代码清单12-6中的配置相比，其与DHCP服务器配置的结果类似。

当内核完成IP地址自动配置后，它就具备了使用提供的参数挂载根文件系统的能力。你可以通过倒数第三行宣布已经挂载NFS根文件系统VFS（virtual file subsystem，虚拟文件子系统）的信息中看出来。在挂载NFS根文件系统后，第5章讲述的内核初始化过程就完成了。

将内核设置为静态IP传递给目标板IP地址，而不是用DHCP或BOOTP服务器为内核获取IP地址，这也是行得通的。可以通过内核命令行直接传递IP地址。这样情况下，内核命令行如下：

```
console=console=ttyS0,115200 \
↳ ip=192.168.1.68:192.168.1.9::255.255.255.0:pdna:eth0:off \
↳ root=/dev/nfs rw nfsroot=192.168.1.9:/home/chris/pdna-target
```

12.4 小结

- 开发环境中的许多特性都能极大地提高嵌入式交叉开发的效率。大多数这些特性都可以归于工具和实用程序之列。下一章介绍开发工具时，将详细探讨这方面的内容。
- 一台配置合理的开发主机对嵌入式开发人员来说是非常关键的资源。
- 交叉平台使用的工具链必须经过正确的配置，以便与主机系统的目标Linux环境相匹配。
- 你的开发主机必须安装目标机组件，这样你的工具链和二进制实用程序才能引用。这些组件包括目标平台的头文件、库文件、目标二进制程序及其相关配置文件。简而言之，你需要装配或获取一份嵌入式Linux发行版。
- 配置诸如TFTP、DHCP和NFS这类的目标服务器可以极大地提高嵌入式Linux开发人员的工作效率。本章用具体的实例逐一介绍了各种服务的配置。

参考资源

GCC 在线文档

<http://gcc.gnu.org/onlinedocs/>

创建和测试gcc/glibc交叉工具链

<http://kegel.com/crosstool/>

TFTP协议，版本2

RFC 1350

www.ietf.org/rfc/rfc1350.txt?number=1350

Bootstrap协议（BOOTP）

RFC 951

www.ietf.org/rfc/rfc0951.txt?number=951

动态主机配置协议

RFC 2131

www.ietf.org/rfc/rfc2131.txt?number=2131



本章内容

- GDB
- DDD
- cbrowser/cscope
- 追踪和程序分析工具
- 二进制实用程序
- 其他二进制实用程序
- 小结

一个典型的嵌入式Linux发行版会包括很多有用的工具。一些工具比较复杂，需要经过大量的练习才能掌握。而另一些则比较简单，它们完成的功能常常会被嵌入式系统开发人员忽略。有些工具可能需要针对特定的环境进行定制。很多工具随手拿来就能够运行，为开发人员提供了有用的信息，而不需要他们付出太多努力。本章讲述的是可以提供给嵌入式Linux工程师的最重要（也是经常忽视）的工具。

本章并不介绍这些工具和实用程序的全部细节，这些内容本身就可以编排为一整本书了。本章不是提供完整的参考，而是为读者介绍每种工具和实用程序的基本用法。建议你继续深入学习这些及其他的重要开发工具，每个工具的帮助手册（或其他文档）都是一个非常好的学习起点。

首先介绍的是GDB，然后简单看看GDB的图形前端DDD（Data Display Debugger）。接下来介绍一系列为开发人员查看程序和系统整体行为而设计的实用程序，包括strace、ltrace、top和ps。对于没有经验的Linux开发人员来说，通常不会注意到这些实用程序。我们随后将介绍一些崩溃转储（crash dump）和内存分析工具。最后介绍一些更为有用的二进制实用程序。

13.1 GDB

如果你花费了大量时间开发Linux应用程序，那么毫无疑问，你一定用了不少时间来熟悉GDB。GDB被认为是开发人员工具箱中最重要的工具。它有着悠久的历史，现在已经发展到具备底层硬件相关的调试的能力，支持大量不同的体系结构和处理器。应该注意，GDB的用户手册几乎和本书篇幅等量。我们在这里提到它，是为了介绍GDB。我建议大家参考本章最后的“参考

资源”提到的用户手册。

因为本书是关于嵌入式Linux开发的，所以我们使用一个交叉编译过的GDB版本。也就是说，调试器自身运行在开发主机上，但却能够解析在编译时配置的体系结构的二进制可执行文件。在后面几个例子中，我们使用Red Hat Linux兼容的开发主机上的GDB，以及XScale（ARM）目标处理器。虽然使用的是gdb缩写形式，但是我们给出的示例是基于XScale的交叉调试器（cross-gdb）。这个调试器是Monta Vista嵌入式Linux发行版的ARM XScale平台附带的。二进制文件名是xscale_be-gdb。它仍然是GDB，只是简单配置成为交叉开发环境使用。

GDB调试器是一个复杂的程序，在构建过程中使用了很多配置选项。提供构建gdb的指南不是我们的本意，其他书中已经谈过了这个话题。根据本章的初衷，我们假定你已经获得了可以工作的GDB，它已经根据你使用的体系结构和主机开发环境进行了配置。

13.1.1 调试核心转储

把GDB从工具箱中拖出来使用的最普遍的原因之一是评估核心转储（core dump）。它快速而又简单，通常能马上找到错误代码。当应用程序出现一个错误时，会产生核心转储，例如内存的非法访问。很多条件会触发核心转储^①，但是SIGSEGV（段错误）是最常见的。SIGSEGV是一个Linux内核信号，它在用户进程产生非法内存访问时生成。当这个信号生成后，内核中止进程，如果内核支持，则会转储一个核心映像。

要产生核心转储，进程必须对资源有所限制。有几种方法可以实现：使用setrlimit()函数调用设置进程的资源限制；在BASH或BusyBox的shell命令提示符下使用ulimit命令。在嵌入式系统的初始化脚本中，看到下面这行命令并不奇怪，这个命令可以使系统在进程产生错误时生成核心转储：

```
$ ulimit -c unlimited
```

BASH内置命令用于设置核心转储的大小限制。在前面的实例中，大小被设置为无限。

当应用程序生成一个段错误（例如，向允许权限外的内存地址写数据）时，Linux中止进程，并产生核心转储（如果开启相应支持的话）。核心转储是一份段错误发生时所运行进程的快照。

它有助于调试二进制程序中的符号。在构建过程中，GDB通过调试符号（gcc-g）生成大量有用的输出。不过，确定导致段错误发生的事件次序还是可能的，即使不使用调试符号编译二进制程序。没有调试符号的协助，你可能需要做一点研究性的工作。你必须手动地把虚拟地址和程序关联到一起。

代码清单13-1列出使用GDB进行核心转储分析会话的输出。输出内容进行了重新编排以适合书的版面。我们使用一些示范软件故意生成一个段错误。下面的内容是某个产生段错误的进程（名为webs）的输出。

```
root@coyote:/workspace/websdemo# ./webs
Segmentation fault (core dumped)
```

① 要了解哪一个信号生成核心转储，请参考.../kernel/signal.c文件中的SIG_KERNEL_COREDUMP_MASK。

代码清单13-1 使用GDB分析核心转储

```

$ xscale_be-gdb webs core
GNU gdb 6.3 (MontaVista 6.3-20.0.22.0501131 2005-07-23)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or distribute cop-
ies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty"
for details.
This GDB was configured as "--host=i686-pc-linux-gnu -target=armv5teb-
↳ montavista-linuxeabi"...

Core was generated by './webs'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /opt/montavista/pro/.../libc.so.6...done.
Loaded symbols for /opt/montavista/pro/.../libc.so.6
Reading symbols from /opt/montavista/pro/.../ld-linux.so.3...done.
Loaded symbols for /opt/montavista/pro/.../ld-linux.so.3
#0 0x00012ac4 in ClearBlock (RealBigBlockPtr=0x0, l=100000000) at led.c:43
43
    *ptr = 0;

(gdb) l
38
39 static int ClearBlock(char * BlockPtr, int l)
40 {
41     char * ptr;
42     for (ptr = BlockPtr; (ptr - BlockPtr) < l; ptr++)
43         *ptr = 0;
44     return 0;
45 }
46 static int InitBlock(char * ptr, int n)
47 {
(gdb) p ptr
$1 = 0x0
(gdb)

```

13.1.2 调用 GDB

代码清单13-1的第一行显示了如何从命令行调用GDB。因为我们进行的是交叉调试，所以需要为我们的主机和目标系统编译的交叉版本的GDB。按上面所示调用交叉版本的gdb，在本例中，传递的二进制文件名是xscale_be-gdb，后面跟着核心转储的文件名，简写为core。打印几行用来描述GDB的配置和其他信息的标语后，GDB打印出中止程序的原因：信号11，这表明是段错误^①。后面几行是GDB载入二进制文件、所依赖的库和核心文件。GDB启动时打印的最后一行是错误发生时程序的当前位置。前面有#0字符串的行意思是栈帧（在虚拟地址0x00012ac4处的ClearBlock()函数中有0号栈帧）。下面一行的行前有一个43，表示led.c文件中发生错误的源

① 信号和相关的号码在Linux内核源码树的.../asm-<arch>/signal.h文件中定义。

代码所在行号。以后，GDB显示它的命令提示符，并等待输入。

为提供一些上下文，我们输入gdb的list命令，使用该命令的缩写形式l。如果没有歧义，GDB可以识别命令的缩写。这里，程序的错误开始于自身。根据GDB对核心转储的分析，错误的行是：

```
43          *ptr = 0;
```

接下来在ptr变量上使用gdb的print命令，同样使用该命令的缩写形式p。如代码清单13-1中所示，指针变量ptr的值是0，所以我们推断段错误的原因是引用了一个空指针，这是一个非常常见的编程错误。在这里，我们推荐使用backtrace命令来查看导致这个错误的调用链，因为这可以带我们返回到实际的错误代码中，代码清单13-2显示了这个结果。

代码清单13-2 backtrace命令

```
(gdb) bt
#0  0x00012ac4 in ClearBlock (RealBigBlockPtr=0x0, l=1000000000) at led.c:43
#1  0x00012b08 in InitBlock (ptr=0x0, n=1000000000) at led.c:48
#2  0x00012b50 in ErrorInHandler (wp=0x325c8, urlPrefix=0x2f648 "/Error",
    webDir=0x2f660 "", arg=0, url=0x34f30 "/Error", path=0x34d68 "/Error",
    query=0x321d8 "") at led.c:61
#3  0x000126cc in webUrlHandlerRequest (wp=0x325c8) at handler.c:273
#4  0x0001f518 in websGetInput (wp=0x325c8, ptext=0xbfffc40,
    pnbytes=0xbfffc38) at webs.c:664
#5  0x0001ede0 in websReadEvent (wp=0x325c8) at webs.c:362
#6  0x0001ed34 in websSocketEvent (sid=1, mask=2, iwp=206280) at webs.c:319
#7  0x00019740 in socketDoEvent (sp=0x34fc8) at sockGen.c:903
#8  0x00019598 in socketProcess (sid=1) at sockGen.c:845
#9  0x00012be8 in main (argc=1, argv=0xbfffe14) at main.c:99
(gdb)
```

backtrace显示返回到用户程序起始处的main()函数的全部调用链。每一行前面都有一个栈帧号。你可以使用gdb的frame命令切换到任何给定的栈帧。代码清单13-3是一个这样的例子。我们在这里切换到栈帧2，显示该帧的源代码。和前面的例子一样，前面有(gdb)的行是向GDB发出的命令，而其他行则是GDB的输出。

代码清单13-3 在GDB中切换栈帧

```
(gdb) frame 2
#2  0x00012b50 in ErrorInHandler (wp=0x325c8, urlPrefix=0x2f648 "/Error",
    webDir=0x2f660 "", arg=0, url=0x34f30 "/Error", path=0x34d68 "/Error",
    query=0x321d8 "") at led.c:61
61          return InitBlock(p, siz);
(gdb) l
56
57          siz = 10000 * sizeof(BigBlock);
58
59          p = malloc(siz);
60          /* if (p) */
```



```

61         return InitBlock(p, siz);
62     /* else return (0); */
63 }
64
65
(gdb)

```

可以看到，使用list命令可以从可用的源代码中获取一点帮助，利用它可以很容易地追踪到使用错误的空指针的代码。实际上，机灵的读者可能已经注意到在本例中产生段错误的源代码了。从代码清单13-3中，我们看到在调用的malloc()函数中，检查返回值的语句被屏蔽了。在本例中，malloc()调用失败，因此导致在调用链的后两帧中，操作指向一个空指针。虽然这个实例是人为的，且意义也不大，但是通过使用GDB和核心转储类似的方法，可以很容易地跟踪很多这种类型的崩溃。你也可以通过查看函数调用中的参数值找到空指针。通常这会直接帮助你找到形成空指针的帧。

13.1.3 GDB 调试会话

我们通过一次典型的调试会话来总结一下GDB。在前面那个崩溃程序示例中，我们采用单步跟踪的方法逐步缩小产生错误的原因。当然，如果有一个核心转储，应该始终从那里开始。不过在其他一些情况下，你可能想设置断点和单步跟踪运行的代码。代码清单13-4详细说明了我們如何为一次调试会话启动GDB。注意，所调试的程序必须是在gcc命令行编译时使用调试标志进行编译的。参考图12-1，这是一个交叉调试会话，在你的开发主机中运行GDB，调试运行在目标板上的程序。我们会在第15章中深入讨论远程应用程序调试的内容。

代码清单13-4 初始化GDB调试会话

```

$ xscale_be-gdb -silent webs

(gdb) target remote 192.168.1.21:2001
0x40000790 in ?? ()
(gdb) b main
Breakpoint 1 at 0x12b74: file main.c, line 78.
(gdb) c
Continuing.

Breakpoint 1, main (argc=1, argv=0xbeffffe04) at main.c:78
78         bopen(NULL, (60 * 1024), B_USE_MALLOCC);
(gdb) b ErrorInHandler
Breakpoint 2 at 0x12b30: file led.c, line 57.
(gdb) c
Continuing.

Breakpoint 2, ErrorInHandler (wp=0x311a0, urlPrefix=0x2f648 "/Error",
webDir=0x2f660 "", arg=0, url=0x31e88 "/Error", path=0x31918 "/Error",
query=0x318e8 "") at led.c:57

```

```

57          siz = 10000 * sizeof(BigBlock);
(gdb) next
59          p = malloc(siz);
(gdb) next
61          return InitBlock(p, siz);
(gdb) p p
$1 = (unsigned char *) 0x0
(gdb) p siz
$2 = 100000000
(gdb)

```

下面来看一下这个简单的调试会话。首先使用gdb target命令连接到目标板，第15章会更详细地讨论远程调试。连接到目标硬件后，使用gdb break（缩写为b）命令在main()函数处设置一个断点，然后使用gdb continue（缩写为c）命令继续执行程序。如果程序有参数，调用GDB时在命令行上直接给出。

设置main()处的断点，执行continue命令（同样使用命令缩写）后，在ErrorInHandler()处再设置一个断点。选中新断点后，使用next命令开始单步跟踪代码。接下来，我们会遇到malloc()函数调用。在malloc()调用以后，检查其返回值后，我们会发现失败了，这是因为返回了空值。最后，打印malloc()调用中参数的值，看到其申请了一个非常大的内存区域（1亿字节），因此会失败。

尽管意义不大，但是本段中的这个GDB示例足以使新手能立刻熟悉GDB。我们中很少有人能真正掌握GDB的用法，因为它实在是太复杂，而且功能非常多。13.2节将介绍GDB的图形前端DDD，这样对不熟悉GDB的人来说会更容易接受。

关于GDB的最后一点提示是，你一定已经注意到了，初次调用GDB时在控制台显示了很多行标语，如代码清单13-1所示。在这些示例中，如前所述，我们使用了来自Monta Vista 嵌入式Linux发行版提供的交叉调试器，标语行包含了嵌入式开发人员必须重视的信息：GDB的主机和目标机的规格。从代码清单13-1中可以看到调用GDB时的输出：

```

This GDB was configured as "--host=i686-pc-linux-gnu -
↳ target=armv5teb-montavista-linuxeabi"

```

在这个例子中，我们调用了在PC机Linux上编译的GDB，即一个运行GNU/Linux操作系统的i686平台。同样重要的是，本例中的GDB也编译为可以调试ARM二进制代码的程序，这类程序通过armv5teb大端工具链生成。

刚接触嵌入式开发的人常犯的一个最常见错误，是在调试目标板上可执行程序时使用了错误的GDB。如果工作不正常，你应该立刻检查GDB的配置，确保其对你的开发环境是合理的。你不能使用本地GDB调试目标板的代码。

13.2 DDD

DDD（数据显示调试器）是GDB和其他命令行调试器的图形前端。除了单纯地查看源代码和在调试会话中单步跟踪以外，DDD有很多高级特性。图13-1是DDD主界面的截图。

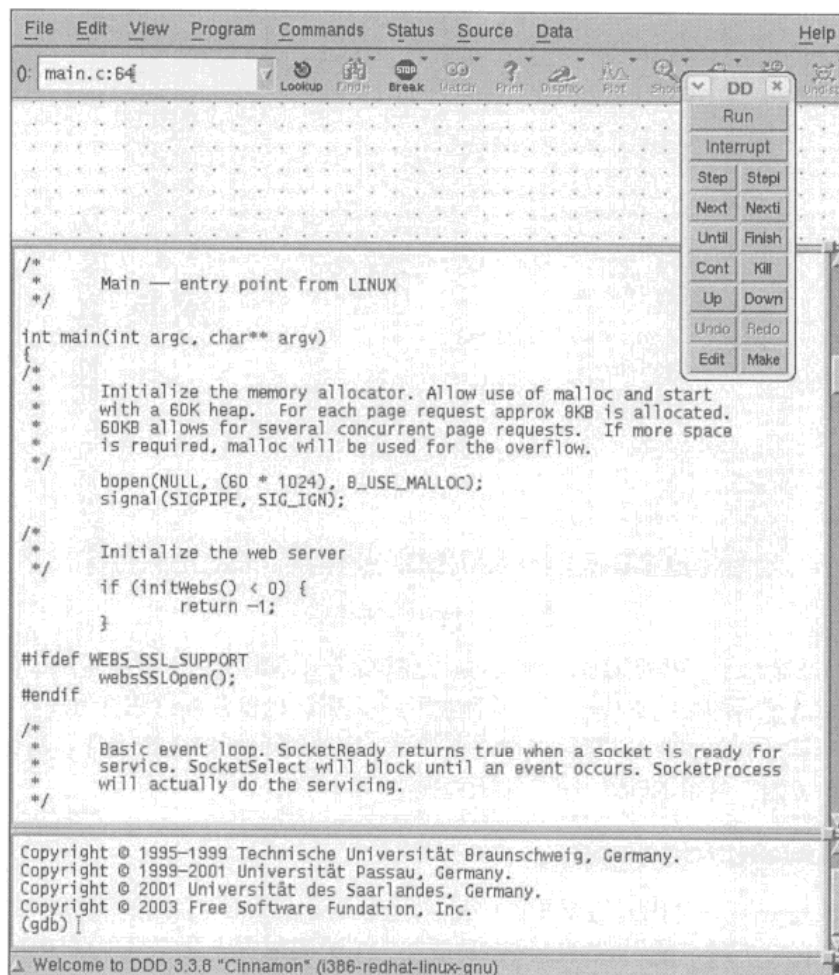


图13-1 DDD（数据显示调试器）

DDD是这样被调用的：

```
$ ddd --debugger xscale_be-gdb webs
```

如果没有--debugger选项，DDD将尝试调用开发主机上的本地GDB。如果你想调试目标板上的应用程序，那么这个GDB不是你需要的。DDD命令行上的第二个参数是要调试的程序。有关DDD的更多内容请翻阅参考手册。

使用图13-1中的命令工具，可以单步跟踪程序。你可以可视化地设置断点，也可以通过DDD窗口下方的GDB控制台窗口进行设置。要调试目标板，必须首先把调试器与目标系统进行连接，如代码清单13-4所示，使用target命令。这条命令在ddd主屏幕的GDB窗口中给出。

连接到目标板后，可以执行前面示例中相似的命令来避免程序运行失败。图13-2给出DDD在这次调试会话中稍后阶段所显示的内容。

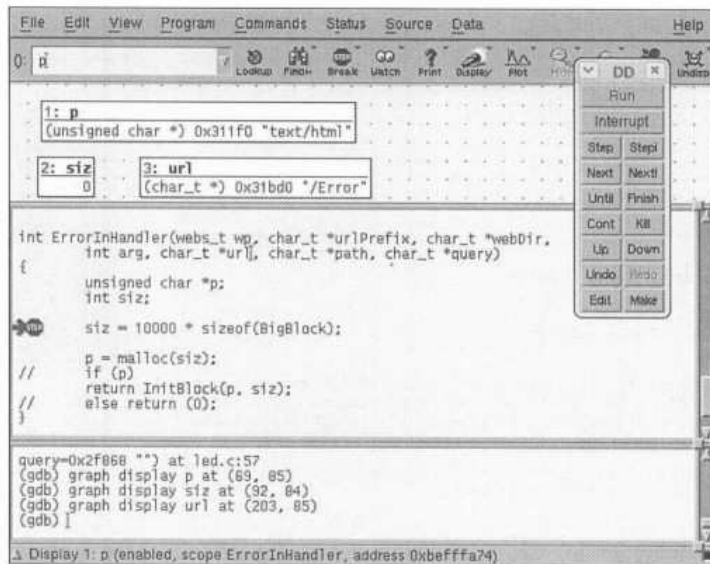


图13-2 DDD调试会话

注意，图13-2显示了一些重要的程序变量，这些变量可以帮助我们逐步排除产生段错误的原因。因为我们可以使用图中的命令工具单步跟踪程序，所以可以监视这些变量。

DDD是一个强大的GDB图形前端。它使用简单，且受多种开发主机支持。具体内容请参考本章后面“参考资源”给出的GNU DDD文档链接。

13.3 cbrowser/cscope

这里提到cbrowser，是因为这个易用的工具很适合阅读Linux内核源码树^①。cbrowser是一个简单的源代码浏览工具，使用它可以很容易地在大型源码树中大量符号中跳转。

Linux内核的makefile支持构建cbrowser使用的数据库。这里是从Linux内核快照提取的一个示例。

```
$ make ARCH=ppc CROSS_COMPILE=ppc_82xx- cscope
```

这条命令会生成cbrowser使用的cscope符号数据库。cscope是引擎，cbrowser是图形用户界面。当然，如果你愿意，可以单独地使用cscope。cscope使用命令行操纵，功能非常强大，但是速度较慢，在这个鼠标流行的时代，不适合在大型源码树中导航。如果vi是你最喜爱的编辑器，那么cscope可能很适合你。

要调用cbrowser，输入包含cscope数据库的目录，简单地输入cbrowser命令即可，不需要任何参数。图13-3给出了一个示例会话。有关这两个有用的工具，可以参考本章后面“参考资源”中列出的内容。

^① 实际上，cbrowser使用的对底层引擎的支持是在Linux构建系统中的。

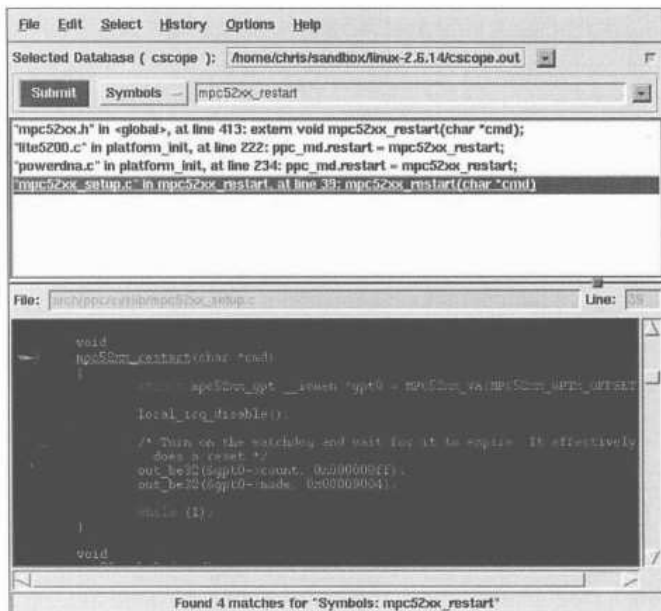


图13-3 cbrowser

13.4 追踪和程序分析工具

有很多有用的工具可以让你看到系统的不同方面。一些工具站在高层观察，例如查看系统中正运行着哪些进程，哪个进程最占用CPU。另一些工具可以提供详细的分析，例如内存存在哪里分配，或更强大一些的，内存存在哪里泄漏。下面几节将介绍这方面的最重要的工具和实用程序。限于篇幅，我们只能粗略介绍这些工具，如果你想了解更详细的内容，请参考相应资料。

13.4.1 strace

实际上，这个强大的系统追踪实用程序在所有Linux发行版中都能找到。strace捕捉并显示Linux应用程序执行的每一个内核系统调用所产生的有用信息。strace随时可用，因为即使没有源代码，也可以在程序中运行。和GDB不同，strace不需要程序带有调试符号。此外，strace是一个非常好的教育工具。正如它的帮助手册中写道的，“学生、黑客和喜欢打破砂锅问到底的人将通过追踪普通的程序，学到非常多的系统和系统调用的知识。”

在准备本章前面GDB内容的示例软件时，我决定使用一个我并不熟悉的软件项目——GoAhead网络服务器的早期版本。第一次尝试编译和链接这个项目时，我发现一个非常有趣的strace的例子。从命令行启动程序，返回控制台，没有产生任何错误消息，查看系统日志也没有任何线索！但它就是不能运行。

strace很快发现了问题。代码清单13-5给出了在这个软件包上调用strace的输出。为节省篇幅，我删除了其中的很多行。未经删减的输出超过一百行。

代码清单13-5 strace输出: GoAhead Web演示

```

01 root@coyote:/home/websdemo$ strace ./websdemo
02 execve("./websdemo", ["/websdemo"], [/* 14 vars */]) = 0
03 uname({sys="Linux", node="coyote", ...}) = 0
04 brk(0) = 0x10031050
05 open("/etc/ld.so.preload", O_RDONLY) = -1 ENOENT (No such file or
↳ directory)
06 open("/etc/ld.so.cache", O_RDONLY) = -1 ENOENT (No such file or
↳ directory)
07 open("/lib/libc.so.6", O_RDONLY) = 3
08 read(3, "\177ELF\1\2\1\0\0\0\0\0\0\0\0\0\3\0\24\0\0\0\1\0\1\322"... , 1024) =
↳ 1024
09 fstat64(0x3, 0x7ffffefc8) = 0
10 mmap(0xfe9f000, 1379388, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0xfe9f000
11 mprotect(0xfffd8000, 97340, PROT_NONE) = 0
12 mmap(0xfffd000, 61440, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_FIXED, 3,
↳ 0x130000) = 0xfffd000
13 mmap(0xffee000, 7228, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|
↳ MAP_ANONYMOUS, -1, 0) = 0xffee000
14 close(3) = 0
15 brk(0) = 0x10031050
16 brk(0x10032050) = 0x10032050
17 brk(0x10033000) = 0x10033000
18 brk(0x10041000) = 0x10041000
19 rt_sigaction(SIGPIPE, {SIG_IGN}, {SIG_DFL}, 8) = 0
20 stat("./umconfig.txt", 0x7ffff9b8) = -1 ENOENT (No such file or
↳ directory)
21 uname({sys="Linux", node="coyote", ...}) = 0
22 gettimeofday({3301, 178955}, NULL) = 0
23 getpid() = 156
24 open("/etc/resolv.conf", O_RDONLY) = 3
25 fstat64(0x3, 0x7ffffd7f8) = 0
26 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
↳ 0x30017000
27 read(3, "#\n# resolv.conf This file is th"... , 4096) = 83
28 read(3, "", 4096) = 0
29 close(3) = 0
... <<< Lines 30-81 removed for brevity
82 socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 3
83 connect(3, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr
↳ ("0.0.0.0")}, 28) = 0
84 send(3, "\267s\1\0\0\1\0\0\0\0\0\0\0\0\6coyotea\0\0\1\0\1", 24, 0) = 24
85 gettimeofday({3301, 549664}, NULL) = 0
86 poll([{fd=3, events=POLLIN, revents=POLLERR}], 1, 5000) = 1
87 ioctl(3, 0x4004667f, 0x7fffe6a8) = 0
88 recvfrom(3, 0x7ffff1f0, 1024, 0, 0x7fffe668, 0x7fffe6ac) = -1 ECONNREFUSED
↳ (Connection refused)
89 close(3) = 0
90 socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 3
91 connect(3, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr
↳ ("0.0.0.0")}, 28) = 0
92 send(3, "\267s\1\0\0\1\0\0\0\0\0\0\0\0\6coyote\0\0\1\0\1", 24, 0) = 24
93 gettimeofday({3301, 552839}, NULL) = 0

```

```

94① poll([{fd=3, events=POLLIN, revents=POLLERR}], 1, 5000) = 1
95 ioctl(3, 0x4004667f, 0x7fffe6a8) = 0
96 recvfrom(3, 0x7ffff1f0, 1024, 0, 0x7fffe668, 0x7fffe6ac) = -1 ECONNREFUSED
↳ (Connection refused)
97 close(3) = 0
98 exit(-1) = ?
99 root@coyote:/home/websdemo#

```

我在strace产生的输出中增加了行号，这样更有可读性。第01行可以看到调用的命令。这里使用了最简单的形式，只是在需要检查的程序前直接加上strace命令。这条命令产生了代码清单13-5中的输出。

这次追踪的每一行都表示websdemo进程对内核的一次系统调用。我们不需要分析和理解每一行的内容，虽然这样做很有好处。我们找的是反常的内容，它们可以查明为什么程序不能运行。在前面几行，Linux建立了程序执行使用的环境。我们看到对/etc/ld.so.*的几个open()系统调用，这个工作是由Linux动态链接器-载入器(ld.so)完成的。实际上，第06行是这个嵌入式开发板没有正确配置的一个线索。此处应该是以前运行ldconfig时产生的链接器缓存。(链接器缓存可以极大地增加搜索共享库引用的速度。)随后可以通过在目标板上运行ldconfig解决这个问题。

从开头直到第19行是最基本的琐事，大多数由载入器和libc初始化。注意程序中第20行要查找一个配置文件，但是没有找到。这可能是我们的软件不运行的重要原因。从第24行开始，程序开始设置和配置它所需要的网络资源。第24~29行打开并读取一个包含DNS服务指令的Linux系统文件以解析主机名。接着到第81行完成本地网络配置活动。大多数这类活动由构建网络基础设施需要的网络设置和配置程序完成。出于简洁性的考虑，我删除了部分代码。

特别要注意网络活动开始的第82行，这里我们让程序试着在全是0的IP地址建立一个TCP/IP连接。再来看看第82行：

```
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 3
```

代码清单13-5中的省略号没什么研究价值。我们不可能知道每一个系统调用的细节，但是可以了解到底发生了什么。socket()系统调用类似于文件系统上的open()调用。在本例中，它的返回值用=号后的值表示，代表一个Linux文件描述符。知道这点后，我们可以把第82~89行的close()系统调用之间的活动和文件描述符3联系起来。

因为我们看到了第88行中出现了“Connection refused (连接被拒绝)”的错误，所以对这组相关的系统调用产生的兴趣。到目前，我们仍然不知道程序为什么不运行，但是这看起来并不正常。我们研究一下第82行的socket()系统调用，它建立了IP通信的端点。第83行很奇怪，因为它试图用全为0的IP地址建立一个远程端点(套接字)的连接。我们不需要是网络专家就能怀疑这可能会是造成麻烦的原因^②。第83行提供了另外一个重要的线索：port参数设置为53。用Google

① 为目标系统创建链接器缓存的内容请参考ldconfig的帮助手册。

② 有时在这个环境中，全0的地址是合适的，不过，我们要研究为什么程序不正常，所以我们应该考虑这个可疑的地方。

快速搜索TCP/IP端口号，得到的结果是端口53是域名服务，即DNS。

第84行又提供了一个线索。我们的开发板的主机名是coyote。在代码清单13-5的第01行的命令提示符中可以看得出来。这次活动看起来是DNS在查找开发板的主机名，却失败了。因为是实验，我们在目标系统的/etc/hosts^①文件中增加了一项，把本地分配的开发板IP地址和本地定义的主机名关联到一起，如下所示：

```
Coyote 192.168.1.21 #The IP address we assigned
```

程序功能开始正常了！虽然我们还没有确切地知道为什么这样会导致程序失败（TCP/IP网络专家可能知道答案），但strace输出让我们看到了DNS查找开发板的名称时出现了错误。当我们纠正了这个错误后，程序正常启动，并打开了服务页面。总结一下，这是一个我们没有源代码可参考的程序，这个二进制映像没有把符号编译进去。使用strace，我们可以确定导致程序失败的原因，并成功地解决了它。

13.4.2 strace 的变体

strace实用程序有很多命令行选项，其中比较有用的一个具有选择追踪系统调用子集的功能。例如，如果你只想看某个进程的与网络相关的活动，使用下面的命令：

```
$ strace -e trace=network process_name
```

这条命令可以产生对所有网络相关的系统调用的追踪，例如socket()、connect()、recvfrom()和send()。这是查看某个程序网络活动的非常有用的方法。还有其他子集。例如，你可以只查看和程序中文件相关的活动，包括open()、close()、read()和write()等。另外的子集包括：进程相关的系统调用、信号相关的系统调用和IPC相关的系统调用。

值得注意的是，strace可以跟踪的程序产生其他进程。调用strace时使用-f选项会指示strace继续跟踪使用fork()系统调用创建的子进程。strace命令还存在无数可能的功能，精通这个强大的实用程序的最好方法就是使用它。这个实用程序及所给出的全部工具，都要特别留意查找并阅读最新的开源文档。在绝大多数Linux主机上执行man strace命令所给出的材料，足够你花一下午的时间去了解。

使用strace的一个非常有用的方法是加-c选项。这个选项将产生应用程序的高层次特征。使用-c选项，strace收集每一个系统调用上的统计信息，它发生了多少次，返回多少次错误，每一个系统调用花费的时间。代码清单13-6是在webs上运行strace -c命令的示例。

代码清单13-6 使用strace进行分析

```
root@coyote$ strace -c ./webs
```

% time	seconds	usecs/call	calls	errors	syscall
29.80	0.034262	189	181		send
18.46	0.021226	1011	21	10	open
14.11	0.016221	130	125		read

① 关于这个系统管理文件请参考man hosts的帮助手册。

11.87	0.013651	506	27	8 stat64
5.88	0.006762	193	35	select
5.28	0.006072	76	80	fcntl64
3.47	0.003994	65	61	time
2.79	0.003205	3205	1	execve
1.71	0.001970	90	22	3 recv
1.62	0.001868	85	22	close
1.61	0.001856	169	11	shutdown
1.38	0.001586	144	11	accept
0.41	0.000470	94	5	mmap2
0.26	0.000301	100	3	mprotect
0.24	0.000281	94	3	brk
0.17	0.000194	194	1	1 access
0.13	0.000150	150	1	lseek
0.12	0.000141	47	3	uname
0.11	0.000132	132	1	listen
0.11	0.000128	128	1	socket
0.09	0.000105	53	2	fstat64
0.08	0.000097	97	1	munmap
0.06	0.000064	64	1	getcwd
0.05	0.000063	63	1	bind
0.05	0.000054	54	1	setsockopt
0.04	0.000048	48	1	rt_sigaction
0.04	0.000046	46	1	gettimeofday
0.03	0.000038	38	1	getpid

100.00	0.114985		624	22 total

了解程序在哪里花费时间，在哪里遇到错误，是非常有用的。有些错误可能是程序运行中正常的一部分，但是有一些可能会占用你不期望被占用的时间。从代码清单13-6中可以看到，使用最长时间的系统调用是execve()，shell用这个调用派生程序。正如你看到的，它只被调用一次。我们看到的另一个有趣的现象是，send()是最频繁使用的系统调用。这对于一个小型网络服务器程序是有意义的。

记住，与这里讨论的其他工具一样，strace必须为你的目标体系结构编译。strace是在你的目标板上而不是开发主机上运行的。必须使用与你的体系结构兼容的版本。如果你购买了商业的嵌入式Linux发行版，那么应该确信这个实用程序已经包含在所选择的体系结构中了。

13.4.3 ltrace

ltrace和strace实用程序紧密相关。ltrace针对库调用，而strace针对系统调用。它们的调用方式类似，命令写在被追踪程序的前面，如下所示：

```
$ ltrace ./example
```

代码清单13-7给出在一个小的示例程序上运行ltrace的输出结果，这个程序执行几个标准C库调用。

代码清单13-7 ltrace示例的输出

```
$ ltrace ./example
```

```

__libc_start_main(0x8048594, 1, 0xbffff944, 0x80486b4, 0x80486fc <unfinished ...>
malloc(256)                                     = 0x804a008
getenv("HOME")                                 = "/home/chris"
strncpy(0x804a008, "/home", 5)                 = 0x804a008
fopen("foo.txt", "w")                          = 0x804a110
printf("$HOME = %s\n", "/home/chris"$HOME = /home/chris
)                                               = 20
fprintf(0x804a110, "$HOME = %s\n", "/home/chris") = 20
fclose(0x804a110)                              = 0
remove("foo.txt")                              = 0
free(0x804a008)                                = <void>
+++ exited (status 0) +++
$

```

对每一个库调用，调用的名字和参数的不同部分将被一起显示出来。类似于strace，然后显示库调用的返回值。同strace一样，这个工具也可以在没有程序源代码的情况下使用。

和strace一样，一些选项可以影响ltrace的行为。可以显示每个库调用上程序计数器的值，这个信息帮助你了解应用程序的程序代码流。和strace一样，可以使用-c选项收集和报告计数、错误和时间统计，使之成为一个简单有效的分析工具。代码清单13-8显示了简单的示例程序使用-c选项后的运行结果。

代码清单13-8 使用ltrace进行记录

```

$ ltrace -c ./example
$HOME = /home/chris

```

% time	seconds	usecs/call	calls	function
24.16	0.000231	231	1	printf
16.53	0.000158	158	1	fclose
16.00	0.000153	153	1	fopen
13.70	0.000131	131	1	malloc
10.67	0.000102	102	1	remove
9.31	0.000089	89	1	fprintf
3.35	0.000032	32	1	getenv
3.14	0.000030	30	1	free
3.14	0.000030	30	1	strncpy
100.00	0.000956		9	total

ltrace工具只对那些在编译时使用了动态链接共享库的程序有效。这是常见的默认情况，除非在编译时明确指定了-static，你可以对二进制文件使用ltrace。还是和strace相似，必须使用针对目标体系结构进行编译的ltrace二进制程序。这些实用程序都在目标板而不是主机开发系统上运行。

13.4.4 ps

可能除了strace和ltrace以外，没有任何实用程序比top和ps更容易被嵌入式系统开发人员忽略了。这两个实用程序都有非常多的选项可用，很轻松就可以用整章的篇幅来讨论这些强大的系统分析工具。几乎每一个嵌入式Linux发行版中都有这些实用程序。

这些实用程序都使用了第9章中介绍到的`/proc`文件系统。如果你懂得如何查找并对查找到的结果进行分析，那么你可以在`/proc`文件系统中找到这些工具传送的大多数信息。这些工具将以易于我们阅读的形式提供这些信息。

`ps`可以列出机器上所有正在运行的进程。它也是非常灵活的，经定制后可提供更强大的数据，这些数据显示了运行机器的状态以及其中正在运行进程的状态。例如，`ps`可以显示每一个进程的调度策略。对于使用实时进程的系统来说，这一功能特别有用。

如果不加选项，`ps`显示调用该命令的用户拥有的全部进程，只有这些进程才和调用命令的终端关联。当用户和终端产生了很多作业后，这个功能就会非常有用。

向`ps`传递选项可能导致`ps`产生混淆，因为`ps`支持相当多的标准（如UNIX的POSIX）以及三种截然不同的选项风格：BSD、UNIX和GNU。通常，BSD选项使用一个或多个字母，没有破折号；UNIX选项使用破折号和字母的组合；GNU使用前面加双破折号的长参数格式。参考`ps`实现的帮助手册了解详细信息。

每个使用`ps`的人可能都有一种喜欢的调用方式，一个特别常用的调用是`ps aux`。该命令显示系统中的每个进程。代码清单13-9是一个嵌入式目标板运行状况的示例。

代码清单13-9 进程列表

```
$ ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.8   1416   508 ?        S      00:00   0:00 init [3]
root         2  0.0  0.0     0     0 ?        S<     00:00   0:00 [ksoftirqd/0]
root         3  0.0  0.0     0     0 ?        S<     00:00   0:00 [desched/0]
root         4  0.0  0.0     0     0 ?        S<     00:00   0:00 [events/0]
root         5  0.0  0.0     0     0 ?        S<     00:00   0:00 [khelper]
root        10  0.0  0.0     0     0 ?        S<     00:00   0:00 [kthread]
root        21  0.0  0.0     0     0 ?        S<     00:00   0:00 [kblockd/0]
root        62  0.0  0.0     0     0 ?        S      00:00   0:00 [pdflush]
root        63  0.0  0.0     0     0 ?        S      00:00   0:00 [pdflush]
root        65  0.0  0.0     0     0 ?        S<     00:00   0:00 [aio/0]
root        36  0.0  0.0     0     0 ?        S      00:00   0:00 [kapmd]
root        64  0.0  0.0     0     0 ?        S      00:00   0:00 [kswapd0]
root       617  0.0  0.0     0     0 ?        S      00:00   0:00 [mtdblockd]
root       638  0.0  0.0     0     0 ?        S      00:00   0:00 [rpciod]
bin        834  0.0  0.7   1568   444 ?        Ss     00:00   0:00 /sbin/portmap
root       861  0.0  0.0     0     0 ?        S      00:00   0:00 [lockd]
root       868  0.0  0.9   1488   596 ?        Ss     00:00   0:00 /sbin/syslogd -r
root       876  0.0  0.7   1416   456 ?        Ss     00:00   0:00 /sbin/klogd -x
root       884  0.0  1.1   1660   700 ?        Ss     00:00   0:00 /usr/sbin/rpc.statd
root       896  0.0  0.9   1668   584 ?        Ss     00:00   0:00 /usr/sbin/inetd
root       909  0.0  2.2   2412  1372 ?        Ss+    00:00   0:00 -bash
telnetd    953  0.3  1.1   1736   732 ?        S      05:58   0:00 in.telnetd
root       954  0.2  2.1   2384  1348 pts/0 Ss     05:58   0:00 -bash
root       960  0.0  1.2   2312   772 pts/0 R+     05:59   0:00 ps aux
```

这是使用`ps`查看输出数据的方式之一。每一列的含义如下：

- `USER`和进程ID（`PID`）列的名称已经给出了解释。
- `%CPU`列表示进程生存期开始后使用CPU的百分比。实际上，CPU的使用率永远也不会累

加到100%。

- %MEM列指示进程驻留的内存与可用总物理内存的比率。
- VSZ列是进程的虚拟内存大小，以千字节为单位。
- RSS列是驻留集大小，并显示一个进程使用的非交换物理内存，也以千字节为单位。
- TTY是进程的控制终端。

本例中大多数进程都没有和控制终端关联。代码清单13-9中使用的ps命令是通过telnet会话发出来的，你可以通过pts/0终端设备看出来。

STAT列描述了在生成这张快照那一刻进程的状态。在这里，s意味着进程处于休眠，它等待某一类事件的发生，通常是I/O。R是指进程处于运行状态（也就是说，如果没有更高的优先级在等待，那么调度器会把CPU的控制权交给该进程）。状态字母后的左括号表示该进程有更高的优先级。

最后一列是进程命令名称。那些在方括号中列出的命令是内核线程。另外还有更多可用的符号和选项，读者可以参考ps的帮助手册。

13.4.5 top

ps只是当前系统在某一时刻的快照，而top则可以给出在一定周期内的系统状态和进程的快照。和ps类似，top有很多命令行和配置选项。它是交互式的，可以在操作过程中重新配置，以此根据特定需求定制显示的内容。

如果不使用选项，top会显示所有正在运行的进程，和代码清单13-9中使用ps aux命令列出进程的方式一样，每3秒钟更新一次。当然，更新时间和top的其他一些选项都是用户可配置的。top屏幕中的前面几行显示系统信息，也是3秒钟更新一次，包括系统更新时间、用户数、进程数及其状态，等等。

代码清单13-10显示了在默认配置下不加任何参数时执行top命令的输出。

代码清单13-10 top

```
top - 06:23:14 up 6:23, 2 users, load average: 0.00, 0.00, 0.00
Tasks: 24 total, 1 running, 23 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.0% us, 0.3% sy, 0.0% ni, 99.7% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 62060k total, 17292k used, 44768k free, 0k buffers
Swap: 0k total, 0k used, 0k free, 11840k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU  %MEM    TIME+  COMMAND
  978 root        16   0  1924   952  780 R   0.3   1.5   0:01.22 top
    1 root        16   0  1416   508  452 S   0.0   0.8   0:00.47 init
    2 root         5 -10    0    0    0 S   0.0   0.0   0:00.00 ksoftirqd/0
    3 root         5 -10    0    0    0 S   0.0   0.0   0:00.00 desched/0
    4 root        -2  -5    0    0    0 S   0.0   0.0   0:00.00 events/0
    5 root        10  -5    0    0    0 S   0.0   0.0   0:00.09 khelper
   10 root        18  -5    0    0    0 S   0.0   0.0   0:00.00 kthread
   21 root        20  -5    0    0    0 S   0.0   0.0   0:00.00 kblockd/0
   62 root        20   0    0    0    0 S   0.0   0.0   0:00.00 pdflush
   63 root        15   0    0    0    0 S   0.0   0.0   0:00.00 pdflush
```

65	root	19	-5	0	0	0	S	0.0	0.0	0:00.00	aio/0
36	root	25	0	0	0	0	S	0.0	0.0	0:00.00	kapmd
64	root	25	0	0	0	0	S	0.0	0.0	0:00.00	kswapd0
617	root	25	0	0	0	0	S	0.0	0.0	0:00.00	mtddblockd
638	root	15	0	0	0	0	S	0.0	0.0	0:00.34	rpciod
834	bin	15	0	1568	444	364	S	0.0	0.7	0:00.00	portmap
861	root	20	0	0	0	0	S	0.0	0.0	0:00.00	lockd
868	root	16	0	1488	596	504	S	0.0	1.0	0:00.11	syslogd
876	root	19	0	1416	456	396	S	0.0	0.7	0:00.00	klogd
884	root	18	0	1660	700	612	S	0.0	1.1	0:00.02	rpc.statd
896	root	16	0	1668	584	504	S	0.0	0.9	0:00.00	inetd
909	root	15	0	2412	1372	1092	S	0.0	2.2	0:00.34	bash
953	telnetd	16	0	1736	736	616	S	0.0	1.2	0:00.27	in.telnetd
954	root	15	0	2384	1348	1096	S	0.0	2.2	0:00.16	bash

代码清单13-10的默认列有PID、用户、进程优先级、进程nice值、进程使用的虚拟内存、驻留的内存、任务使用的共享内存数和其他与前面ps实例中描述的一样的列。

限于篇幅，这里只能粗略地介绍这些实用程序。建议你花时间认真阅读top和ps的帮助手册。

13.4.6 mtrace

mtrace软件包是一个分析和报告应用程序中malloc()、realloc()和free()调用的简单实用程序。它易于使用，能帮助你找出应用程序中潜在的问题。与本章提到的其他用户空间实用程序一样，必须针对你的体系结构配置和编译mtrace。mtrace是安装在你的目标机上的一个替代malloc的库。通过一个特殊的函数调用，你的应用程序也可以使用。你的嵌入式Linux发行版中应该包含mtrace软件包。

为了示范这个实用程序的用法，我们编写了一个在简单的链接列表上动态生成数据的程序。每一个列表项都是动态生成的，数据则放在列表里。代码清单13-11给出了这个简单的列表结构。

代码清单13-11 简单的线性链接列表

```
struct blist_s {
    struct blist_s *next;
    char *data_item;
    int item_size;
    int index;
};
```

每个列表项都使用malloc()动态创建，然后顺序放在列表最后：

```
struct blist_s *p = malloc( sizeof(struct blist_s) );
```

列表中的数据项也是动态生成的，这些数据项的容量可变，在列表项放置到列表最后之前，数据会加到列表项中。这样，创建每一个列表项都调用malloc()两次。一次是列表项本身，如代码中的struct blist_s，还有一次是这些可变的数据项。接着在列表中生成10 000条包含可变字符串数据的记录，所以调用malloc()20 000次。

要使用mtrace，必须满足下面三个条件：

- 源代码文件中必须包含mcheck.h头文件；

□ 应用程序必须调用mtrace(), 以安装处理程序;

□ MALLOC_TRACE环境变量必须指定一个可写文件的文件名, 用于记录所追踪的数据。

满足这些条件后, 追踪函数的每一次调用都会在MALLOC_TRACE定义的原始追踪文件中增加一行。所追踪的数据看起来如:

```
@ ./mt_ex:[0x80486ec] + 0x804a5f8 0x10
```

@号表示追踪行内包含地址或者函数名。在前面的例子中, 程序在0x80486ec地址处执行。使用二进制实用程序或调试器, 可以很容易地把这个地址和函数关联在一起。加号(+)是指这是一个分配内存的调用, 而free()调用则用减号指明。下一列指出分配或释放的内存地址所对应的虚拟地址。最后一列是所有分配内存调用的总数。

这种数据格式不太友好。正因如此, mtrace软件包中还包括另一个用来分析原始追踪数据, 并报告问题所在工具^①。举个最简单的例子, 用Perl脚本打印一行消息: “无内存泄漏”。代码清单13-12给出检测到内存泄漏时的输出。

代码清单13-12 mtrace错误报告

```
$ mtrace ./mt_ex mtrace.log
```

```
Memory not freed:
```

```
-----
Address      Size      Caller
0x0804aa70   0x0a     at /home/chris/temp/mt_ex.c:64
0x0804abc0   0x10     at /home/chris/temp/mt_ex.c:26
0x0804ac60   0x10     at /home/chris/temp/mt_ex.c:26
0x0804acc8   0x0a     at /home/chris/temp/mt_ex.c:64
```

可以看到, 这个简单的工具可以在错误发生前或发生时帮助你发现问题。注意, 这个Perl脚本显示了文件名和每一次调用malloc()的行号, 但对于给定的内存地址却没有相应的free()调用。这需要可执行程序中的调试信息才能显示, 调试信息是在编译时传递给编译器-g选项生成的。如果没有调试信息, 脚本只报告调用malloc()函数的地址。

13.4.7 dmalloc

dmalloc可以完成mtrace做不到的事情。mtrace软件包比较简单, 多用于检测mallo/free不配对的情况。dmalloc软件包能够发现更多的动态内存管理方面的错误。与mtrace相比较, dmalloc与源代码结合更紧密。根据配置的不同, dmalloc可以减慢应用程序运行的速度。如果你因为竞争条件或其他时序的问题而怀疑内存错误, 那dmalloc绝对不是你需要的工具。dmalloc(还有mtrace等几个工具)确实能改变应用程序的时序。

dmalloc是一个非常强大的动态内存分析工具。它的可配置性很强, 因此显得有些复杂。学习和掌握这个工具需要多花些功夫。不过, 从QA(质量评价)测试到bug排除, 它都会是你最需要的开发工具。

^① 分析工具是一个mtrace软件包支持的Perl脚本。

dmalloc是一个调试malloc库的替换品。使用dmalloc必须满足下面一些条件:

- 应用程序代码必须含有dmalloc.h头文件;
- 应用程序必须链接dmalloc库;
- dmalloc库和实用程序必须安装在嵌入式目标板上;
- 应用程序在目标机上运行以前, dmalloc库引用的某些环境变量必须定义好。

虽然不是严格需要, 但是你应该在应用程序中加入dmalloc.h头文件, 这会使dmalloc在输出时显示文件和行号。

将应用程序链接到你选择的dmalloc库上。根据你在配置软件包时的选择, dmalloc软件包可以被配置为生成几个不同的库。在下面的例子中, 我们选择使用libdmalloc.so共享库对象。库(或链接的对象)还应放在编译器能找到的地方。编译应用程序的命令可能会是:

```
$ ppc_82xx-gcc -g -Wall -o mtest_ex -L../dmalloc-5.4.2/ \
-lmalloc mtest_ex.c
```

这条命令假定你已经把dmalloc库(libdmalloc.so)放在了通过命令行中-L开关搜索的位置里, 也就是../dmalloc-5.4.2。

把dmalloc库安装到目标板上, 放在你喜欢的位置(估计是/usr/local/lib)。你还要配置系统以便能找到这个库。在PowerPC系统实例中, 我们在/etc/ld.so.conf文件中增加了/usr/local/lib路径, 并调用ldconfig工具更新库搜索缓存。

准备工作的最后一个步骤是, 设置一个dmalloc库使用的环境变量, 该变量可以决定调试级别。环境变量中包含一个调试位掩码, 它把很多功能合并到一个变量中, 看起来可能是这样:

```
DMALLOC_OPTIONS=debug=0x4f4ed03,interv=100,log=dmalloc.log
```

这里, debug是调试级别位掩码, interv设置dmalloc库执行自身和堆的详尽检查的时间间隔。dmalloc库会把日志输出到log变量指定的文件中。

dmalloc软件包里有一个实用程序可以根据传递给它的选项生成DMALLOC_OPTIONS环境变量。上面的例子就是用下面的dmalloc调用产生的。dmalloc软件包中的文档非常全面地介绍了这部分内容, 这里不再赘述。

```
$ dmalloc -p check-fence -l dmalloc.log -i 100 high
```

完成这些步骤后, 你应该能依靠dmalloc调试库运行应用程序了。

dmalloc会产生非常详细的输出日志。代码清单13-13给出一个dmalloc日志输出示例, 这个例子程序中故意生成一些内存泄漏。

代码清单13-13 dmalloc的日志输出

```
2592: 4002: Dmalloc version '5.4.2' from 'http://dmalloc.com/'
2592: 4002: flags = 0x4f4e503, logfile 'dmalloc.log'
2592: 4002: interval = 100, addr = 0, seen # = 0, limit = 0
2592: 4002: starting time = 2592
2592: 4002: process pid = 442
2592: 4002: Dumping Chunk Statistics:
2592: 4002: basic-block 4096 bytes, alignment 8 bytes
```

```

2592: 4002: heap address range: 0x30015000 to 0x3004f000, 237568 bytes
2592: 4002:   user blocks: 18 blocks, 73652 bytes (38%)
2592: 4002:   admin blocks: 29 blocks, 118784 bytes (61%)
2592: 4002:   total blocks: 47 blocks, 192512 bytes
2592: 4002: heap checked 41
2592: 4002: alloc calls: malloc 2003, calloc 0, realloc 0, free 1999
2592: 4002: alloc calls: realloc 0, memalign 0, valloc 0
2592: 4002: alloc calls: new 0, delete 0
2592: 4002:   current memory in use: 52 bytes (4 pnts)
2592: 4002:   total memory allocated: 27546 bytes (2003 pnts)
2592: 4002:   max in use at one time: 27546 bytes (2003 pnts)
2592: 4002:   max allocated with 1 call: 376 bytes
2592: 4002:   max unused memory space: 37542 bytes (57%)
2592: 4002: top 10 allocations:
2592: 4002:   total-size  count in-use-size  count  source
2592: 4002:         16000    1000         32     2 mtest_ex.c:36
2592: 4002:         10890    1000         20     2 mtest_ex.c:74
2592: 4002:          256      1         0      0 mtest_ex.c:154
2592: 4002:         27146    2001         52     4 Total of 3
2592: 4002: Dumping Not-Freed Pointers Changed Since Start:
2592: 4002: not freed: '0x300204e8|s1' (10 bytes) from 'mtest_ex.c:74'
2592: 4002: not freed: '0x30020588|s1' (16 bytes) from 'mtest_ex.c:36'
2592: 4002: not freed: '0x30020688|s1' (16 bytes) from 'mtest_ex.c:36'
2592: 4002: not freed: '0x300208a8|s1' (10 bytes) from 'mtest_ex.c:74'
2592: 4002:   total-size  count  source
2592: 4002:         32      2 mtest_ex.c:36
2592: 4002:         20      2 mtest_ex.c:74
2592: 4002:         52      4 Total of 2
2592: 4002: ending time = 2592, elapsed since start = 0:00:00

```

有一点很重要，这个日志是在程序退出时生成的。（`dmalloc`有很多选项和操作模式，也可以配置成检测到错误就打印输出行）。

输出日志的前一半报告了应用程序中堆和所有内存使用的统计信息。总数是由每一个 `malloc` 库调用产生的，例如 `malloc()`、`free()` 和 `realloc()`。令人感兴趣的是，这份日志报告了最大的10次内存分配以及每次分配所对应的源代码的位置。这对系统级的分析是非常有用的。

在日志末端，我们看到了应用程序中内存泄漏的证据。你可以看到 `dmalloc` 库检测到有4处已分配的内存显然没有被释放。因为包括了 `dmalloc.h` 文件，并在编译时使用了调试符号，所以分配内存所在的源代码的位置也记录到了日志中。

和本章中介绍其他工具一样，限于篇幅，这里只能对这个异常强大的调试工具进行简要的介绍。`dmalloc` 可以检测很多其他情况和限制。例如，`dmalloc` 可以检测什么时候一个释放过的指针被使用。它能告诉你一个用来访问数据的指针是否越界，是否在应用程序许可地址范围内。实际上，`dmalloc` 可以配置为记录差不多任何使用 `malloc` 类调用完成的内存事务。`dmalloc` 绝对是一个值得你花时间了解的好工具。

13.4.8 内核 oops

虽然不是严格意义上的工具，但内核oops（kernel oops）包含了大量有用的信息，可以帮助

你检修错误。内核oops由多种内核错误产生，从由进程产生的简单的内存错误（大多数情况下可完全恢复），到严重的内核崩溃。最近的Linux内核除了支持原始的十六进制地址的值以外，还支持符号信息的显示。代码清单13-14给出一个PowerPC目标板上的内核oops。

代码清单13-14 内核oops

```
$ modprobe loop
Oops: kernel access of bad area, sig: 11 [#1]
NIP: C000D058 LR: C0085650 SP: C7787E80 REGS: c7787dd0 TRAP: 0300 Not tainted
MSR: 00009032 EE: 1 PR: 0 FP: 0 ME: 1 IR/DR: 11
DAR: 00000000, DSISR: 22000000
TASK = c7d187b0[323] 'modprobe' THREAD: c7786000
Last syscall: 128
GPR00: 0000006C C7787E80 C7D187B0 00000000 C7CD25CC FFFFFFFF 00000000 80808081
GPR08: 00000001 C034AD80 C036D41C C034AD80 C0335AB0 1001E3C0 00000000 00000000
GPR16: 00000000 00000000 00000000 100170D8 100013E0 C9040000 C903DFD8 C9040000
GPR24: 00000000 C9040000 C9040000 00000940 C778A000 C7CD25C0 C7CD25C0 C7CD25CC
NIP [c000d058] strcpy+0x10/0x1c
LR [c0085650] register_disk+0xec/0xf0
Call trace:
[c00e170c] add_disk+0x58/0x74
[c90061e0] loop_init+0x1e0/0x430 [loop]
[c002fc90] sys_init_module+0x1f4/0x2e0
[c00040a0] ret_from_syscall+0x0/0x44
Segmentation fault
```

注意，寄存器转储在适当的地方包括了符号信息。要使这些符号信息可用，内核必须开启KALLSYMS。图13-4显示了General Setup主菜单下的配置选项。

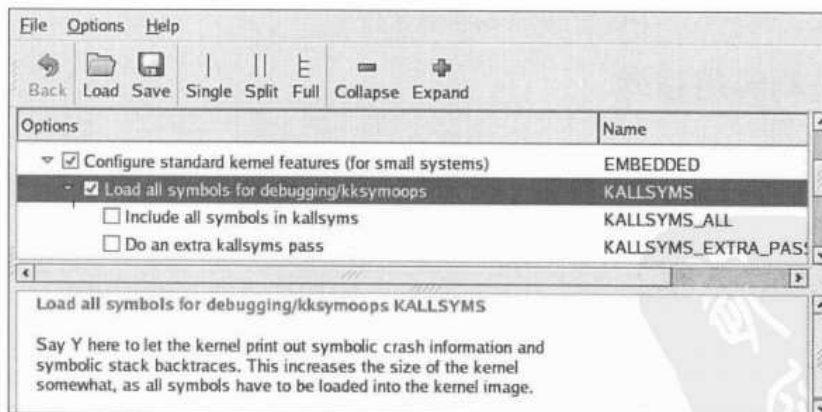


图13-4 oops的符号

内核oops消息中的大多数信息都和处理器紧密相关。要彻底理解oops消息，需要懂得体系结构底层的知识。

分析代码清单13-14中的oops，可以立刻看到，oops是由于“kernel access of bad area, sig: 11”产生的。我们已经从本章前面的例子中知道，信号11是段错误。

代码的第1段是有关oops产生原因、几个重要的指针和出问题任务的概要说明。在代码清单13-14中，NIP是下一条指令的指针，在后面的oops消息中解码。它指向错误的并导致oops的代码。LR是一个PowerPC寄存器，通常用来指示当前执行的子程序的返回地址。SP是栈指针。REGS指包含寄存器转储数据的数据结构所在的内核地址。TARP指示这个oops消息相关的异常种类。参考第7章中最后给出的PowerPC体系结构参考手册，可以看到TRAP 0300是PowerPC数据存储中断，它在数据内存访问错误时被触发。

在oops消息的第3行，可以看到另一个PowerPC寄存器，比如MSR（机器状态寄存器）和它的一些位的解码。下一行是DAR（数据访问寄存器），这个寄存器一般包含错误内存地址。DSISR寄存器中的内容可以和PowerPC体系结构参考手册一道用来找出异常的确切原因。

一条oops消息还包含了任务指针和解码的任务名，以快速确定oops出现时正在运行哪个进程或线程。我们还看到一个详细的处理器寄存器转储，它能带来额外的线索。同样，我们需要了解体系结构和编译器寄存器使用的知识，这样才能从寄存器的值中找到线索。例如，PowerPC体系结构使用r3寄存器保存C函数的返回值。

如果内核支持符号，那么在oops消息的最后一部分将提供带有符号解码的栈轨迹。通过这个消息，我们可以解释一连串导致问题产生的原因。

在这个简单的例子中，我们学习了很多来自oops消息的内容。我们知道了，是一个错误的数据内存访问（与内存取指访问相反）导致了PowerPC数据存储异常。DAR寄存器告诉我们产生这个异常的数据地址在0x0000_0000。我们知道是modprobe进程产生了错误。通过向后追踪（backtrace）和NIP（下一条指示指针），我们更定位到在调用strcpy()的过程中，返回到loop.ko模块中的loop_init()函数时的问题：modprobe试图在发生异常的时候插入这个模块。有了这个信息，追踪错误的空指针引用的代码应该非常容易了。

13.5 二进制实用程序

二进制实用程序（即binutils）是工具链的关键组成部分。实际上，为构建编译器，必须首先成功地构建binutils。在本节中，我们简要介绍嵌入式开发人员需要了解的一些比较有用的工具。和本章介绍的大部分其他工具一样，这些都是交叉工具，而且必须构建为在开发主机上运行，所处理的文件是所选择的体系结构的目标机上的二进制文件。你也可以自己编译这套工具，也可以直接获取到需要的版本。不过，这里为这些示例假定了一个交叉开发环境。

13.5.1 readelf

Readelf实用程序检查目标ELF二进制文件的组成。这个对于构建ROM或闪存中的映像特别有用，我们可能需要清楚地了解映像文件的布局。readelf同样还是一个学习工具链如何构建映像，以及理解ELF文件格式的好工具。

例如，要显示一个ELF映像的符号表，使用这条命令：

```
$ readelf -s <elf-image>
```

要显示ELF映像的全部字段，使用这条命令：

```
$ readelf -e <elf-image>
```

使用-s选项可以列出ELF映像的段头。看到一个仅有7行的“hello world”程序居然包含了38个单独的段，你可能会感到很惊讶。也许其中的几段你很熟悉，如.text和.data段。代码清单13-15包含了取自“hello world”程序的部分代码。简洁起见，我们只列出了嵌入式开发人员熟悉或相关的段。

代码清单13-15 readelf段头

```
$ ppc_82xx-readelf -S hello-ex
There are 38 section headers, starting at offset 0x32f4:

Section Headers:
[ Nr] Name           Type           Addr          Off           Size       ES Flg Lk Inf Al
...
[11] .text              PROGBITS       100002f0 0002f0 000568 00 AX 0 0 4
...
[13] .rodata             PROGBITS       10000878 000878 000068 00 A 0 0 4
...
[15] .data               PROGBITS       100108e0 0008e0 00000c 00 WA 0 0 4
...
[22] .sdata              PROGBITS       100109e0 0009e0 00001c 00 WA 0 0 4
[23] .sbss               NOBITS         100109fc 0009fc 000000 00 WA 0 0 1
...
[25] .bss                NOBITS         10010a74 0009fc 00001c 00 WA 0 0 4
...
```

.text段包含可执行程序代码，.rodata段包含程序中的常量，.data段一般包含C库的序言（prologue）代码使用的已初始化的全局数据。.data段还可以包含应用程序中大容量的已初始化的数据。.sdata段用于保存较小容量的已初始化的全局数据，.sdata只在一些体系结构中才会存在。一些处理器体系结构可以在已知内存区域的属性已知的情况下，利用已优化过的数据进行访问。.sdata和.sbss段可以进行这类的优化。.bss和.sbss段包含程序中未初始化的数据。这些段不占用程序映像空间，在程序启动后，C库的序言代码为它们分配内存空间，并初始化为0。

我们可以转储这些段，显示其内容。在C程序的所有函数外给定这行，可以查看.rodata段中是如何布局的：

```
char *hello_rodata = "This is a read-only data string\n";
```

使用readelf命令可以指定想要转储的段号，如代码清单13-15所示：

```
$ ppc_82xx-readelf -x 13 hello-ex
Hex dump of section '.rodata':
0x10000878 100189e0 10000488 1000050c 1000058c .....
0x10000888 00020001 54686973 20697320 61207265 ....This is a read-
0x10000898 61642d6f 6e6c7920 64617461 20737472 only data string
0x100008a8 696e670a 00000000 54686973 20697320 .....This is
0x100008b8 73746174 69632064 6174610a 00000000 static data.....
0x100008c8 48656c6c 6f20456d 62656464 65640a00 Hello Embedded..
0x100008d8 25730a00 25780a00          %s..%x..
```

我们看到了声明过的全局变量在.rodata段中出现，同时还有程序中所有的常量字符串。

13.5.2 使用 readelf 检查调试信息

readelf一个更有用的特性是显示ELF文件中的调试信息。在编译时使用了-g选项，编译器会在生成的ELF文件中产生几个段的调试信息。我们可以使用readelf显示ELF文件中的这些ELF段头：

```
$ ppc-linux-readelf -S ex_sync | grep debug
[28] .debug_aranges PROGBITS 00000000 000c38 0000b8 00 0 0 8
[29] .debug_pubnames PROGBITS 00000000 000cf0 00007a 00 0 0 1
[30] .debug_info PROGBITS 00000000 000d6a 00079b 00 0 0 1
[31] .debug_abbrev PROGBITS 00000000 001505 000207 00 0 0 1
[32] .debug_line PROGBITS 00000000 00170c 000354 00 0 0 1
[33] .debug_frame PROGBITS 00000000 001a60 000080 00 0 0 4
[34] .debug_str PROGBITS 00000000 001ae0 00014d 00 0 0 1
```

给readelf加上--debug-dump选项，可以显示任何一个.debug_*段的内容。在第14章讨论调试已优化内核代码时，你会发现这些信息非常有用。

调试信息可能会非常多。显示Linux内核ELF文件vmlinux的全部调试信息将输出600多万行信息。不过，尽管看起来令人恐惧，但熟悉少许调试信息将使你成为一名更为合格的嵌入式工程师。

代码清单13-16是从一个小示例程序中取到的.debug_info段的部分内容。为节省篇幅，我们只列出部分记录。

代码清单13-16 部分调试信息转储

```
$ ppc-linux-readelf --debug-dump=info ex_sync
1 The section .debug_info contains:
2
3   Compilation Unit @ 0:
4     Length:      109
5     Version:      2
6     Abbrev Offset: 0
7     Pointer Size:  4
8   <0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
9     DW_AT_stmt_list : 0
10    DW_AT_low_pc      : 0x10000368
11    DW_AT_high_pc     : 0x1000038c
12    DW_AT_name        :
13    ..../sysdeps/powerpc/powerpc32/elf/start.S
14    DW_AT_comp_dir     : /var/tmp/BUILD/glibc-2.3.3/csu
15    DW_AT_producer     : GNU AS 2.15.94
16    DW_AT_language     : 32769 (MIPS assembler)
17    ...
394  <1><5a1>: Abbrev Number: 14 (DW_TAG_subprogram)
395    DW_AT_sibling      : <5fa>
396    DW_AT_external     : 1
397    DW_AT_name         : main
398    DW_AT_decl_file    : 1
```

```

399     DW_AT_decl_line   : 9
400     DW_AT_prototyped  : 1
401     DW_AT_type        : <248>
402     DW_AT_low_pc      : 0x100004b8
403     DW_AT_high_pc     : 0x10000570
404     DW_AT_frame_base  : 1 byte block: 6f      (DW_OP_reg31)
...
423 <2><5e9>: Abbrev Number: 16 (DW_TAG_variable)
424     DW_AT_name       : mybuf
425     DW_AT_decl_file   : 1
426     DW_AT_decl_line   : 11
427     DW_AT_type        : <600>
428     DW_AT_location    : 2 byte block: 91 20   (DW_OP_fbreg: 32)
...

```

通过Dwarf2[®]标签DW_TAG_compile_unit识别的第一条记录，确定这个PowerPC可执行程序的第一个编译单元，这里是start.S文件，该文件为C程序提供一段启动程序。通过DW_TAG_subprogram识别的下一条记录确定用户程序的起始点，即我们熟悉的main()函数。这条Dwarf2调试记录包含对这个文件的引用以及main()函数所在的行号。代码清单13-16中的最后一个记录确定main()函数中的一个本地变量，这个程序是mybuf。同样，记录中包含所在的行号和文件。你可以从这些信息中推断出main()函数在第9行，而mybuf在源文件中第11行。ELF文件中的其他调试记录通过Dwarf2 DW_AT_decl_file属性与文件名关联。

可以参考本章结尾的“参考资源”了解Dwarf2调试信息格式的全部内容。

13.5.3 objdump

objdump工具一般会与readelf交迭使用。不过，objdump的一个强大特性是显示反汇编目标代码的能力。代码清单13-17给出了PowerPC版本的“hello world”程序的.text段反汇编代码示例。为节省篇幅，这里只包括main()函数。包括C库的序言和结语在内的完整转储将占用很多篇幅。

代码清单13-17 使用objdump反汇编

```

$ ppc_82xx-objdump -S -m powerpc:common -j .text hello
...
10000488 <main>:
10000488:    94 21 ff e0      stwu    r1,-32(r1)
1000048c:    7c 08 02 a6      mflr    r0
10000490:    93 e1 00 1c      stw     r31,28(r1)
10000494:    90 01 00 24      stw     r0,36(r1)
10000498:    7c 3f 0b 78      mr      r31,r1
1000049c:    90 7f 00 08      stw     r3,8(r31)
100004a0:    90 9f 00 0c      stw     r4,12(r31)
100004a4:    3d 20 10 00      lis     r9,4096
100004a8:    38 69 08 54      addi    r3,r9,2132
100004ac:    4c c6 31 82      crclr   4*cr1+eq

```

① 本章最后给出了Dwarf2调试信息规范参考资料。

```

100004b0:      48 01 05 11      bl      100109c0

<__bss_start+0x60>
100004b4:      38 00 00 00      li      r0,0
100004b8:      7c 03 03 78      mr      r3,r0
100004bc:      81 61 00 00      lwz     r11,0(r1)
100004c0:      80 0b 00 04      lwz     r0,4(r11)
100004c4:      7c 08 03 a6      mtlr    r0
100004c8:      83 eb ff fc      lwz     r31,-4(r11)
100004cc:      7d 61 5b 78      mr      r1,r11
100004d0:      4e 80 00 20      blr
...

```

这个简单的main()函数的大多数代码是栈帧的创建和销毁。真正对printf()函数的调用由代码中间附近0x100004b0地址处的跳转指令(bl)完成。这是一个PowerPC函数调用。因为这个程序被编译为动态链接的对象,所以我们在它链接到共享库printf()程序前不能获得printf()函数的地址。一旦我们把它编译为静态链接对象,就能看到调用printf()函数的符号和相应的地址了。

13.5.4 objcopy

objcopy能够格式化二进制目标文件,以及可选择地转换二进制目标文件格式。这个工具对于生成保存在ROM或闪存中的映像文件的代码非常有用。在第7章中介绍的U-Boot就利用objcopy把最终的ELF文件生成了二进制和s-record^①两种输出格式。下面示例的用法解释了objcopy的能力和如何构建闪存映像的用法。

```
$ ppc_82xx-objcopy --gap-fill=0xff -O binary u-boot u-boot.bin
```

objcopy说明一个映像可能是为闪存而准备的。在本例中,输入文件u-boot是完整的ELF格式的U-Boot映像,它包含了符号和重定向信息。objcopy工具只把包括程序代码和数据相关的段放在输出的文件映像中,这里指定为u-boot.bin。

擦除过的闪存中包含的都是1。因此,用1填满一个二进制映像可以提高程序效率并延长闪存的寿命。这项功能可以使用objcopy的--gap-fill参数完成。

这只是objcopy用法的一个简单举例。这个工具可以生成s-records格式,并在格式间进行转换。更多内容请参考帮助手册。

13.6 其他二进制实用程序

在你的工具链中还包含另外几个有用的实用程序,学习使用这些实用程序是比较容易的。你会发现这些工具很实用。

13.6.1 strip

strip可以用来剥除(strip)一个二进制文件中的符号和调试信息。这个实用程序常用于节

^① s-record 文件是二进制文件的ASCII表示,很多设备程序开发人员和软件二进制实用程序都要使用。

省嵌入式设备上的空间。在一个交叉开发的模型中，把剥除后的二进制程序放在目标系统上，把未处理的版本放在开发主机中会很方便。使用这种方法，文件中剩余的符号可用于在开发主机上的交叉调试，而又节省了目标板上的空间。strip有很多选项，在它的帮助手册中都有说明。

13.6.2 addr2line

当提到代码清单13-12中的mtrace时，你会看到来自mtrace分析脚本的输出，其中包含了文件和所在行号的信息。mtrace Perl脚本使用addr2line实用程序读取可执行ELF文件中的调试信息，并显示对应地址所在行。运行同一个mtrace示例，给定一个虚拟地址，我们可以找到文件名及所在行号。

```
$ addr2line -f -e mt_ex 0x80487c6
put_data
/home/chris/examples/mt_ex.c:64
```

注意，put_data()函数也与文件名和行号一同被列出。也就是说，地址0x80487c6运行着mt_ex.c文件的第64行的put_data()函数。对于比较大的二进制文件，这个实用程序更有用，通常这些文件由多个文件组成，例如Linux内核：

```
$ ppc_82xx-addr2line -f -e vmlinux c000d95c
mpc52xx_restart
arch/ppc/syslib/mpc52xx_setup.c:41
```

这个特殊的例子强调了本章多次重复的要点：这是一个和体系结构相关的工具。你必须使用配置和编译为与你的目标体系结构相匹配的工具。和交叉编译器一样，addr2line是一个交叉工具，是二进制实用程序包的一部分。

13.6.3 strings

strings实用程序可以查看二进制文件中的ASCII字符串数据。对于当源代码或调试符号不可用时，使用strings检查内存转储特别有用。你可能经常发现，通过跟踪反编译的二进制文件的字符串，可以查找程序崩溃的原因。虽然strings有几个命令行选项，但是很容易学习和使用。更多细节内容请参考帮助手册。

13.6.4 ldd

虽然不是严格意义上的二进制实用程序，但对于嵌入式开发人员来说，ldd脚本是另一个非常有用的工具。它是C库的一部分，基本上在每一个Linux发行版中都有。ldd可以列出给定目标文件或文件所依赖的共享库。第11章中介绍了ldd，具体使用方法参考代码清单11-2。ldd脚本在开发ramdisk映像时非常有用。在各种嵌入式Linux邮件列表中，一个最常见的问题是挂载根系统后出现的内核崩溃：

```
VFS: Mounted root (nfs filesystem).
Freeing unused kernel memory: 96k init
Kernel panic - not syncing: No init found. Try passing init=option to kernel.
```

其中一个最常见的原因是根文件系统映像（可能是ramdisk、闪存或NFS根文件系统）不支持内核试图执行的二进制程序调用的库。使用ldd可以确定每一个二进制程序需要的库，并确保在ramdisk或其他根文件系统映像里包括了它们。在前面的内核崩溃的例子中，init确实包含在文件系统中，但是没有Linux动态加载器ld.so.1。ldd的使用方法相当简单：

```
$ xscale_be-ldd init
libc.so.6 => /opt/mvl/.../lib/libc.so.6 (0xdead1000)
ld-linux.so.3 => /opt/mvl/.../lib/ld-linux.so.3 (0xdead2000)
```

这个简单的例子证实了init二进制程序需要两个动态库对象：libc和ld-linux。这两个库必须存放在目标板上，而且init二进制程序必须有权限访问，即它们必须是可读和可执行的。

13.6.5 nm

nm实用程序显示一个目标文件中的符号。在多种任务中，这可能很有用。例如，交叉编译一个大应用程序时，你遇到了未解析的符号（unresolved symbol）错误，这时可以使用nm查找是哪一个目标模块包含哪些符号，并修改构建环境以便包括它。

nm实用程序提供了每一个符号的属性。例如，你可以看到这个符号是本地的还是全局的，或它是事先被定义的还是从一个特定的目标模块引用的。代码清单13-18给出在U-Boot ELF映像上运行nm时的几行输出结果：

代码清单13-18 使用nm显示符号

```
$ ppc_85xx-nm u-boot
...
fff23140 b base_address
fff24c98 B BootFile
fff06d64 T BootpRequest
fff00118 t boot_warm
fff21010 d border
fff23000 A __bss_start
...
```

注意这些U-Boot符号的链接地址。它们被链接到闪存设备上，这块开发板的闪存已经被映射到内存的最高处。出于学习的角度，这个代码清单只包含了几个符号。中间一列是符号类型。大写字母表示全局符号，小写字母表示本地符号。字母B表示符号在.bss段，T表示符号在.text段，D表示符号在.data段，A表示一个绝对地址，不能被其他链接过程修改。绝对符号表示.bss段的开始，由在启动时清除.bss的代码使用，通常C执行环境需要这段代码。

13.6.6 prelink

prelink实用程序通常在一些非常强调启动时间的系统中使用。动态链接的ELF可执行文件，当首次加载程序时，该文件必须在运行时先被链接。对于大应用程序来说，这个时间会很长。prelink准备了共享库和对象文件，并根据它们提供对未解析的库引用的相关内容。实际上，这可以减少应用程序的启动时间。其帮助手册给出了这个程序的全部用法。

13.7 小结

- GDB是一个复杂而强大功能的调试器，我们介绍了其基本内容以便引导读者入门。
- GDB的图形前端是DDD，它整合了源代码和GDB的命令行接口的数据显示能力。
- cbrowser是一个对于理解大型工程有用的工具。它使用cscope数据库快速查找和显示C源码中的符号以及其他组成成分。
- 很多追踪和程序分析工具支持Linux。我们介绍了几个，包括strace、ltrace、top和ps，以及内存分析类工具mtrace和dmalloc。
- 嵌入式开发人员通常需要构建定制的映像，例如那些引导装入程序和固件映像所需要的映像。要完成这些任务，掌握binutils的知识是不可缺少的。我们介绍了binutils中的很多实用程序，包括readelf、objdump、objcopy和其他实用程序。

参考资源

GDB: GNU工程调试器

www.gnu.org/software/gdb/gdb.html

GDB 袖珍参考

Arnold Robbins

O'Reilly Media, 2005

数据显示调试器:

www.gnu.org/software/ddd/

cbrowser主页:

<http://cbrowser.sourceforge.net/>

cscope主页:

<http://cscope.sourceforge.net/index.html>

dmalloc: Malloc调试库:

<http://dmalloc.com/>

工具接口标准 (TIS): 可执行和链接格式 (ELF) 规范1.2版

TIS委员会, 1995年5月

工具接口标准:

DWARF 调试信息格式规范2.0版

TIS委员会, 1995年5月

本章内容

- 内核调试的难点
- 使用KGDB调试内核
- Linux内核的调试
- 硬件辅助调试
- 无法启动时
- 小结

项目的进度会受开发人员查找和修改项目bug的效率所影响。特别是在内核开发过程中，内核调试非常繁琐，不管用什么办法调试内核，这个过程都是非常复杂的。本章将详细介绍内核调试的方法，并讲述内核和设备驱动程序的调试技巧，提高调试效率。

14.1 内核调试的难点

调试一个现代操作系统通常面临很多挑战，尤其在采用了虚拟内存的操作系统上进行调试具有更独特的挑战。那种用在线调试器代替处理器进行调试的时代已经过去了，处理器的运行速度越来越快，功能也越来越复杂。而且，流水线的体系结构技术隐藏了很多重要的代码执行的细节。部分原因是总线上访问内存的顺序和代码执行的顺序并不一致，这是因为处理器的内部缓存存放了指令流。不可能将外部总线的访问次序和内部处理器的指令执行次序对应起来，除非是从一个相当粗的层次上来看。

在调试Linux内核过程中面临的挑战主要有：

- 出于执行效率的原因，Linux内核源代码中的许多地方进行了高度优化。
- 编译器将C源代码编译成机器指令时，使用了大量优化技术，使机器指令变得更加复杂。内联函数就是这种优化的一个例子。
- 因为编译器对代码的优化，单步调试代码的时候通常会产生和期望不一样的结果。
- 虚拟内存技术将内核空间 and 用户空间隔离开来，在调试的时候上下文要发生切换，难于调试。
- 使用传统的调试办法，有些代码不能单步跟踪。

- 启动代码尤其不容易调试，因为在启动阶段全是硬件相关的代码，而且可以使用的资源非常有限，比如没有串口支持、没有建立内存映射等。

Linux内核已经成为一个高性能、成熟稳定的操作系统，相对其他商业操作系统一点也不逊色。内核里面的很多模块如果只通过阅读代码的方式，会很难理解。要理解内核模块中代码，必须了解体系结构和详细设计。有几本详细描述内核设计的书写得非常好，读者可以参考本章“参考资源”中的书目。

GCC是一个具备优化代码功能的编译器。默认情况下，Linux内核采用-O2级别的优化选项进行编译。这样一来，很多代码在优化编译后，不能和原先的结构、控制流等保持一致^①。比如，Linux内核使用了大量的内联函数。内联函数用关键字inline声明，编译时，内联函数的代码直接包含在最终的执行线程中^②，不会产生函数调用的指令。内联函数要求-O1以上的优化级别，因此编译内核的时候不能禁止优化选项，尽管关掉优化会让调试更容易。

在对Linux内核的很多地方进行调试时，单步跟踪代码是非常困难并且不可能实现的。最明显的例子是跟踪修改虚拟内存设置的代码路径。当应用程序调用到系统调用，该系统调用通过int 80中断进入内核，这将导致进程能看到的地址空间发生变化。事实上，任何引起处理器对运行上下文的变化，调试起来都非常困难，几乎不可能单步跟踪调试。

14.2 使用 KGDB 调试内核

在对Linux内核进行源代码级别调试的时候，有两种常用的方法：

- 使用KGDB作为远程gdb代理；
- 使用硬件JTAG调试器来探测并控制处理器。

14.4节将介绍JTAG调试方法。

KGDB（内核GDB）是一套Linux内核的补丁，通过远程串行协议提供gdb的接口。KGDB在目标机上实现一个gdb的桩，于是主机上运行的交叉gdb就能够和目标机通信。一直以来，目标机上运行的KGDB都要求必须通过串口线和主机连接起来。目前所知，最新的KGDB版本才能支持通过以太网的连接进行内核调试。官方网站www.kernel.org上的内核还没有完全支持KGDB。所以要使用KGDB调试，必须要将KGDB移植到内核，或者从其他途径取得一个，专门针对所选择的体系结构和目标平台，并且要求已经包含了KGDB支持的嵌入式Linux发行版。

图14-1是KGDB调试建立的过程。目标板要和主机建立三个物理连接。以太网连接用于目标系统加载NFS根文件系统，并且可以使用主机通过telnet登录到目标机上。如果目标系统在闪存中有一个ramdisk的映像，目标板将ramdisk启动挂载作为根文件系统，就可以将以太网的连接去掉。

串口用于连接目标系统上的KGDB和主机上运行的gdb，可选的第二个串口作为一个控制终端。如果目标系统只有一个串口，那么调试KGDB起来就会比较麻烦。

① 查看GCC的用户手册，在本章“参考资源”中有关于优化级别详细的文档。

② 内联函数的功能和宏比较相似，它的优点是编译时会做安全类型检查。

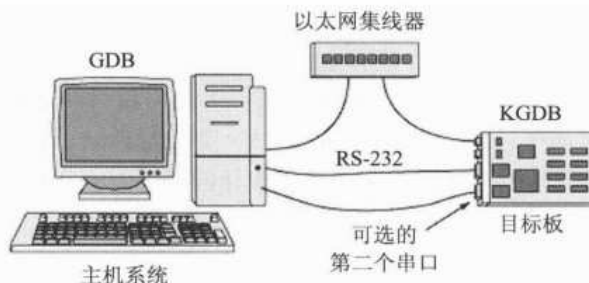


图14-1 KGDB调试建立过程

如图14-1所示，交叉调试器gdb运行在开发主机上，KGDB是目标系统启动后内核的一部分。KGDB实现主机上gdb远程调试目标板的钩子功能，用来作为主机上的gdb接口，可以通过操作gdb来实现调试时候的功能，比如设置断点、检查内存、开启程序单步调试运行等。

14.2.1 KGDB 内核配置

KGDB是一个内核编译选项，在编译内核时必须将该选项选中。KGDB出现在内核的Kernel Hacking菜单中，如图14-2所示。作为内核配置的一部分，必须选择KGDB使用的串行端口。从图14-2可以看到，我们选中了“Compile the kernel with debug info”选项。一旦选中，编译内核的时候就会加上-g编译选项，使内核支持符号调试。

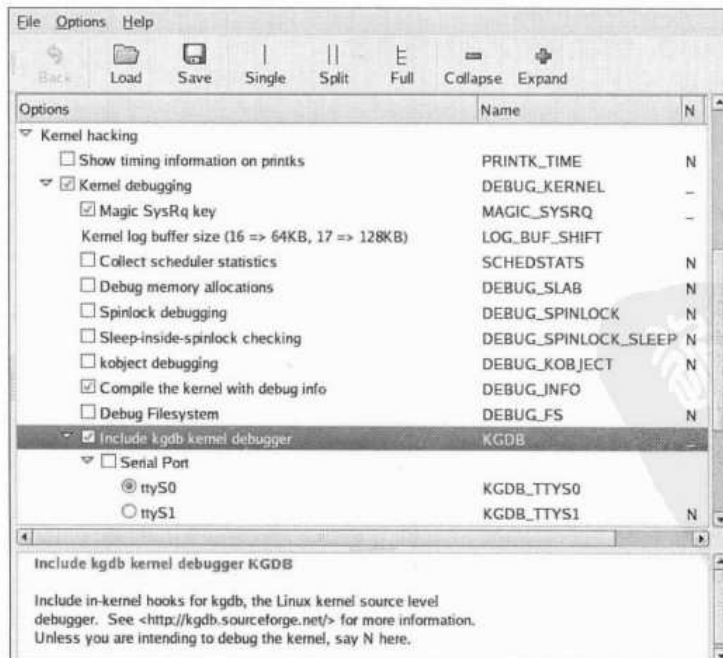


图14-2 内核中的KGDB配置

14.2.2 支持 KGDB 的内核启动

编译完成支持KGDB的内核后，必须开启该功能（KGDB）。但是，各种不同的目标系统和平台使KGDB开启的方法各不相同。通常采用的办法是通过内核参数传递一个命令行开关。如果KGDB被编译进内核，但是内核参数没有加上使KGDB生效的开关，那么KGDB将什么都不做。当开启KGDB功能后，内核将在启动阶段早期就停在KGDB-enabled断点处，这个断点处即目标系统等待主机通过gdb连接到目标板。图14-3显示了KGDB最开始设置断点的逻辑流程图。

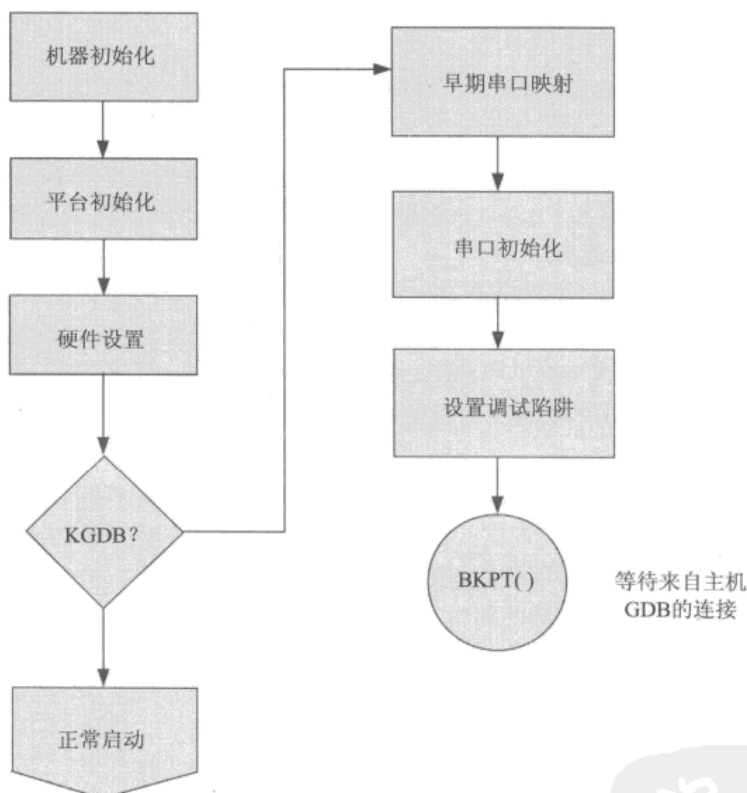


图14-3 KGDB逻辑

KGDB需要通过串口连接到主机^①。所以在KGDB流程中，首先要在启动阶段建立对串口的支持。在很多体系结构中，串口硬件在访问前必须被映射到内存中。当地址范围被映射后，接下来就是初始化串口，然后初始化调试陷阱（debugtrap），使CPU在遇到调试异常时候可以通过查找陷阱来进行异常处理。

代码清单14-1显示了KGDB功能开启的内核启动后，终端的输出情况。该例子是基于硬件平台AMCC 440EP（Yosemite 板），引导装入程序采用U-Boot。

^① 当然，现在也可以通过以太网进行KGDB调试了。

代码清单14-1 使用U-Boot启动支持KGDB功能的内核

```

=> sete bootargs console=ttyS1,115200 root=/dev/nfs rw ip=dhcp gdb
=> bootm 200000
## Booting image at 00200000 ...
Image Name:      Linux-2.6.13
Image Type:      PowerPC Linux Kernel Image (gzip compressed)
Data Size:       1064790 Bytes = 1 MB
Load Address:    00000000
Entry Point:     00000000
Verifying Checksum ... OK
Uncompressing Kernel Image ... OK
$T0440:c000ae5c;01:c0205fa0;#d9 <<< See text

```

大部分的启动序列和第7章介绍的一致。这个启动过程和前面介绍的启动过程有两点不同：内核参数里面有开启KGDB的选项，内核解压缩之后会打印出一段看上去很奇怪的文字。

回忆第7章介绍的U-Boot，它可以通过环境变量bootargs设置内核命令参数。在那个例子中，我们在bootargs参数设置中加入了gdb，该参数就让内核在启动开始阶段设置断点，并且让内核停下来等待主机的交叉调试器连接。

如图14-3所示，内核发现gdb参数的存在，就会将控制权交给主机上的gdb调试器。通过代码清单14-1打印出来的文本ASCII字符序列可以验证该点。如果要研究这个gdb远程串行协议，可以参考本章最后的参考资源。在这个例子中，KGDB发送一个Stop Reply数据包给远程主机上的gdb，告诉gdb断点陷阱的信息。例子中的两个32位的参数c000ae5c和c0205fa0分别标识程序的位置和栈帧的位置。

现在内核已经建立起来并且等待主机的调试，我们就可以正式开始调试会话了。通过在主机上调用交叉gdb连接到目标机上进行调试。在本例中，我们共享同一个串口来进行，所以在调用gdb连接目标机之前要先断开先前的终端连接。代码清单14-2是gdb连接的过程。假定之前已经退出终端仿真器（在U-Boot上启动内核的终端，代码清单14-1所示），并将该终端用于gdb调试。

代码清单14-2 连接到KGDB

```

$ ppc_4xx-gdb --silent vmlinux
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
breakinst () at arch/ppc/kernel/ppc-stub.c:825
825     }
(gdb) l
820         return;
821     }
822
823     asm("    .globl breakinst    \n\
824         breakinst: .long 0x7d821008");
825 }
826
827 #ifdef CONFIG_KGDB_CONSOLE
828 /* Output string in GDB O-packet format if GDB has connected.
If nothing
829     output, returns 0 (caller must then handle output). */
(gdb)

```

本例子中依次输入如下3个命令：

- 调用gdb，将ELF格式的内核vmlinux传给gdb；
- 在gdb里使用target remote命令连接到目标机；
- 使用list命令的缩写形式l，显示当前调试对应于源代码中的位置。

很显然，传给gdb调试用的vmlinux内核映像必须和目标板上运行的二进制格式的内核是同一个编译配置编译出来的内核。为了让vmlinux带有调试信息，必须在编译的时候使用-g编译选项。

当输入target remote命令后，gdb将打印出当前程序指令所指的位置（在内核源代码中的具体位置）。本例中，内核在源代码文件.../arch/ppc/kernel/ppc-stub.c的第825行断点设置处停下来，此处的代码是用汇编代码写的。当调试重新开始之后，将继续从825行开始执行。

14.2.3 有用的内核断点

我们现在已经建立了一个和目标板的调试连接。当在gdb提示符下输入continue(c)命令，内核继续执行，如果内核启动过程不发生错误，启动过程将完成。在很多体系结构和处理器上使用KGDB有一个很小的限制，因此我们必须要在KGDB设计中作出取舍：在内核启动初期需要支持内核调试（例如，在一个完整的支持中断的串口驱动安装前），但是要兼顾KGDB调试工具的功能的完备，我们需要让KGDB易于使用、稳定且容易移植。当内核在运行的时候，KGDB使用一个简单的驱动程序轮询串口设备。这样做的一个弊端是，使用传统的Ctrl+C或Break序列来终止进程将失去效果。因此，如果在调试的过程中要停止正在运行的内核，只能用断点的方法，或者让内核运行时遇到异常。

出于上面讲到的原因，在一些全局范围的地方定义断点非常重要，能够暂停当前正在执行的内核线程。代码清单14-3是两个在内核中设置断点的最常用方法。

代码清单14-3 常用的内核断点

```
(gdb) b panic
Breakpoint 1 at 0xc0016b18: file kernel/panic.c, line 74.
(gdb) b sys_sync
Breakpoint 2 at 0xc005a8c8: file fs/buffer.c, line 296.
(gdb)
```

使用gdb breakpoint命令（本例中使用它的缩略版本的命令b），我们设置了两处断点。一个断点在panic()函数处，另外一个在系统调用sys_sync()处。这样可以使内核在遇到panic之前停下来检测它的状态。当在目标机上调用sync从用户空间进入内核空间的时候，第二个断点用于暂停内核，并且进入调试状态，这是一个非常有效的调试方法。

现在我们继续调试会话。目标板上已经有一个开启了KGDB功能的内核正在运行，内核停在KGDB定义断点处。通过主机的交叉调试器连接到目标机。本例中，通过ppc_4xx-gdb调用建立调试会话，并且设置了两个有用的系统断点。现在我们就可以着手开始内核的调试。

提示：文件.../arch/ppc/setup.c的功能是在KGDB获得系统控制权之前完成系统的早期

初始化工作，在建立目标机的KGDB和主机的交叉gdb连接之前，我们不能使用断点调试。图14-3显示了KGDB在获得控制权之前的大致过程。调试KGDB之前的内核启动过程要通过硬件调试。14.4节将讲述如何使用硬件调试内核。

14.3 Linux 内核的调试

需要单步调试跟踪内核的一个常见的原因是，要修改和调试开发板相关的代码。现在让我们看看如何调试AMCC Yosemite开发板。首先在体系结构相关代码的setup函数处设置断点，然后让开发板一直运行到断点处。代码清单14-4显示了该过程。

代码清单14-4 调试体系结构相关的代码

```
(gdb) b yosemite_setup_arch
Breakpoint 3 at 0xc021a488:
file arch/ppc/platforms/4xx/yosemite.c, line 308.
(gdb) c
Continuing.
Can't send signals to this remote system. SIGILL not sent.

Breakpoint 3, yosemite_setup_arch () at arch/ppc/platforms/4xx/yosemite.c:308
308         yosemite_set_emacdata();
(gdb) l
303     }
304
305     static void __init
306     yosemite_setup_arch(void)
307     {
308         yosemite_set_emacdata();
309
310         ibm440gx_get_clocks(&clocks, YOSEMITE_SYSCLK, 6 * 1843200);
311         ocp_sys_info.opb_bus_freq = clocks.opb;
312
(gdb)
```

当执行到yosemite_setup_arch()函数断点处时，gdb显示此时对应于源代码yosemite.c的第308行处，控制权转交给gdb。第一个l命令列出源代码第308行的上下文。在continue(c)命令后面跟着的警告消息可以安全地忽略掉。这其实是gdb测试远程系统性能的一个方法。它首先发送远程continue_with_signal命令到目标机。但是该目标板的KGDB不支持这个命令，于是该命令被目标系统丢弃。gdb将不能发送到目标板的命令信息打印出来，显示该条信息“Can't send signals to this remote system. SIGILL not sent.”之后等待进一步的调试。

14.3.1 gdb 远程串口协议

gdb包含一个可以查看串口协议的调试开关，开启该选项可以观察主机和目标机之间的远程串口协议。这不但对理解底层协议非常有帮助，而且可以解决目标系统出现异常时遇到的问题。下面的命令是开启该调试模式的：


```
(gdb) set debug remote 1
```

开启该调试模式之后，对gdb中continue命令引起的动作序列进行观测非常有指导意义。代码清单14-5是continue命令的引起的动作序列。

代码清单14-5 远程协议中的continue例子

```
(gdb) c
Continuing.
Sending packet: $mc0000000,4#80...Ack
Packet received: c022d200
Sending packet: $Mc0000000,4:7d821008#68...Ack
Packet received: OK
Sending packet: $mc0016de8,4#f8...Ack
Packet received: 38600001
Sending packet: $Mc0016de8,4:7d821008#e0...Ack
Packet received: OK
Sending packet: $mc005bd5c,4#23...Ack
Packet received: 38600001
Sending packet: $Mc005bd5c,4:7d821008#0b...Ack
Packet received: OK
Sending packet: $mc021a488,4#c8...Ack
Packet received: 4bffffbad
Sending packet: $Mc021a488,4:7d821008#b0...Ack
Packet received: OK
Sending packet: $c#63...Ack
<<< program running, gdb waiting for event
```

乍一看，虽然有些吓人，但是却很容易理解这个动作序列代表什么。总的说来，gdb正在恢复目标系统上的所有断点信息。回顾代码清单14-3，我们输入了两个断点，一个在panic()，另一个在sys_sync()。在后面的代码清单14-4中，我们将在函数yosemite_setup_arch()处增加第三个断点。这些断点信息可以在gdb提示符下面输入info breakpoints命令查看。通常情况下，我们使用简写版本：i b。

```
(gdb) i b
Num Type           Disp Enb Address          What
1  breakpoint      keep y   0xc0016de8 in panic at kernel/panic.c:74
2  breakpoint      keep y   0xc005bd5c in sys_sync at fs/buffer.c:296
3  breakpoint      keep y   0xc021a488 in yosemite_setup_arch at
arch/ppc/platforms/4xx/yosemite.c:308
breakpoint already hit 1 time
(gdb)
```

现在来比较在前面设置断点的地址信息和代码清单14-5中的gdb远程\$mc包中的地址信息。\$m包是一个读取目标内存的命令，\$M包是一个写目标内存的命令。对每个断点都有一次这样的过程：断点的地址从目标内存中读出，然后用gdb存储在主机，并且重置PowerPC的trap指令twge r2, r2 (0x7d821008)，该指令将导致控制权传递给调试器。图14-4解释了这个动作。

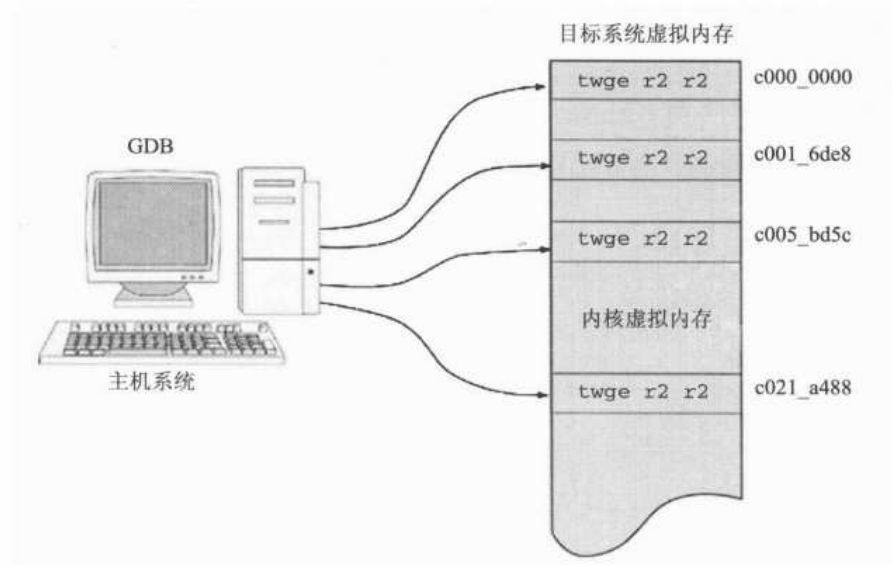


图14-4 插入目标内存断点

你可能从图中已经注意到gdb正在更新四个断点，但是我们只输入了三个断点信息。第一个断点在目标内存位置0xc000_0000处，这是gdb在启动阶段自动加上的。这个位置是内核在链接成最终的ELF文件时的内核的入口地址_start。这就好比在用户空间调试程序时在main()设置断点一样，只不过该过程是gdb自动完成的。另外三个断点是我们先前自己设置的。

当将控制权交给gdb时，同样的事情按照相反的次序发生。代码清单14-6显示了当执行到yosemite_setup_arch()断点处时的引发的动作序列细节。

代码清单14-6 远程协议：遇到断点

```

Packet received: T0440:c021a488;01:c020ff90;
Sending packet: $mc0000000,4#80...Ack <<< Read memory @c0000000
Packet received: 7d821008
Sending packet: $Mc0000000,4:c022d200#87...Ack <<< Write memory
Packet received: OK
Sending packet: $mc0016de8,4#f8...Ack
Packet received: 7d821008
Sending packet: $Mc0016de8,4:38600001#a4...Ack
Packet received: OK
Sending packet: $mc005bd5c,4#23...Ack
Packet received: 7d821008
Sending packet: $Mc005bd5c,4:38600001#cf...Ack
Packet received: OK
Sending packet: $mc021a488,4#c8...Ack
Packet received: 7d821008
Sending packet: $Mc021a488,4:4bffffbad#d1...Ack
Packet received: OK

Sending packet: $mc021a484,c#f3...Ack
Packet received: 900100244bffffbad3fa0c022

```

```
Breakpoint 3, yosemite_setup_arch () at arch/ppc/platforms/4xx/yosemite.c:308
308      yosemite_set_emacdata();
(gdb)
```

\$T包是一个gdb停止响应包。当遇到断点的时候，它被目标系统发送到gdb。在我们的例子中，\$T包返回了程序指令寄存器的值和寄存器r1的值^①。该活动的其他部分和代码清单14-5显示的是相反的过程。PowerPC陷阱断点的指令被删除掉，gdb会把原始指令恢复到它们各自所在的内存位置。

14.3.2 调试优化后的内核代码

在本章开始，我们曾说过内核调试代码中的一个挑战是因为编译器优化代码所致。我们知道，Linux内核在默认情况下是用优化级别-O2编译的。因此，我们以后使用-O1优化级别来编译内核简化调试任务。我们将讲述多种优化方法中的一个，以此表明优化是如何使调试内核过程变得复杂化的。

相关的因特网邮件列表充满了对使用一些糟糕的工具的问题报告。有时发帖人报告说他的调试工具支持单步调试，但是它的行号不能和源代码匹配。这里我们通过一个例子说明编译器的优化是如何让源代码级的调试变复杂的。在这个例子中，当遇到断点时，gdb报告的行号并不能和我们源文件中的内联函数的行号一致。

为了演示该实例，我们使用和代码清单14-4中相同的调试代码。但是在本例中，我们在编译内核时，使用-O2编译选项，这是Linux内核的默认编译选项。代码清单14-7显示了这次调试会话过程中打印的结果。

代码清单14-7 优化后的体系结构相关的代码

```
$ ppc_44x-gdb --silent vmlinux
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
breakinst () at arch/ppc/kernel/ppc-stub.c:825
825      }
(gdb) b panic
Breakpoint 1 at 0xc0016b18: file kernel/panic.c, line 74.
(gdb) b sys_sync
Breakpoint 2 at 0xc005a8c8: file fs/buffer.c, line 296.
(gdb) b yosemite_setup_arch
Breakpoint 3 at 0xc020f438: file arch/ppc/platforms/4xx/yosemite.c, line 116.
(gdb) c
Continuing.

Breakpoint 3, yosemite_setup_arch ()
at arch/ppc/platforms/4xx/yosemite.c:116

116      def = ocp_get_one_device(OCF_VENDOR_IBM, OCF_FUNC_EMAC, 0);
(gdb) l
111      struct ocp_def *def;
```

① 我们之前指出，gdb远程协议的细节在gdb手册中有详细介绍，该手册在本章末的“参考资源”。

```

112         struct ocp_func_emac_data *emacdata;
113
114         /* Set mac_addr and phy mode for each EMAC */
115
116         def = ocp_get_one_device(OCP_VENDOR_IBM, OCP_FUNC_EMAC, 0);
117         emacdata = def->additions;
118         memcpy(emacdata->mac_addr, __res.bi_enetaddr, 6);
119         emacdata->phy_mode = PHY_MODE_RMII;
120
(gdb) p yosemite_setup_arch
$1 = {void (void)} 0xc020f41c <yosemite_setup_arch>

```

参考代码清单14-4，可以知道函数yosemite_setup_arch()实际上在文件yosemite.c的第306行。但是代码清单14-7中明确显示，当我们遇到断点后，gdb报告文件yosemite.c的断点所在的行号是第116行。初次看上去好像是调试器和相应的源代码发生了误匹配。这是gdb的bug吗？首先，让我们确保编译器产生了调试符号信息。利用在第13章中所讲的readelf^①工具，我们检查编译器对该函数产生的调试信息。

```

$ ppc_44x-readelf --debug-dump=info vmlinux | grep -u6 \
yosemite_setup_arch | tail -n 7
DW_AT_name      : (indirect string, offset: 0x9c04): yosemite_setup_arch
DW_AT_decl_file : 1
DW_AT_decl_line : 307
DW_AT_prototyped : 1
DW_AT_low_pc    : 0xc020f41c
DW_AT_high_pc   : 0xc020f794
DW_AT_frame_base : 1 byte block: 51      (DW_OP_reg1)

```

要认识在源代码文件中第307行所遇到的问题，我们并不需要成为研究DWARF2调试记录的专家^②，只需要利用addr2line实用程序（也是在第13章中介绍）来研究这一点。将代码清单14-7中从gdb中分离出来的地址（最后一行的0xc020f41c）在addr2line实用程序中使用。

```

$ ppc_44x-addr2line -e vmlinux 0xc020f41c
arch/ppc/platforms/4xx/yosemite.c:307

```

很明显，上面的结果显示断点是在源代码的第307行。然而，gdb却报告设置的断点在源代码文件yosemite.c中的第116行。为了理解此过程究竟发生了什么，需要查看该函数产生的汇编代码。代码清单14-8是gdb对yosemite_setup_arch()函数调用反汇编命令disassemble后输出的汇编结果。

代码清单14-8 反汇编yosemite_setup_arch()函数

```

(gdb) disassemble yosemite_setup_arch
0xc020f41c <yosemite_setup_arch+0>:    mflr    r0
0xc020f420 <yosemite_setup_arch+4>:    stwu    r1,-48(r1)
0xc020f424 <yosemite_setup_arch+8>:    li      r4,512
0xc020f428 <yosemite_setup_arch+12>:   li      r5,0

```

- ① 记住使用交叉版本的readelf工具，对PowerPC 44x体系结构平台使用ppc_44x-readelf工具。
- ② 对DWARF调试规范的参考本章末“参考资源”中的材料。

```

0xc020f42c <yosemite_setup_arch+16>:  li    r3,4116
0xc020f430 <yosemite_setup_arch+20>:  stmw   r25,20(r1)
0xc020f434 <yosemite_setup_arch+24>:  stw    r0,52(r1)
0xc020f438 <yosemite_setup_arch+28>:  bl     0xc000d344
<ocp_get_one_device>
0xc020f43c <yosemite_setup_arch+32>:  lwz    r31,32(r3)
0xc020f440 <yosemite_setup_arch+36>:  lis    r4,-16350
0xc020f444 <yosemite_setup_arch+40>:  li     r28,2
0xc020f448 <yosemite_setup_arch+44>:  addi   r4,r4,21460
0xc020f44c <yosemite_setup_arch+48>:  li     r5,6
0xc020f450 <yosemite_setup_arch+52>:  lis    r29,-16350
0xc020f454 <yosemite_setup_arch+56>:  addi   r3,r31,48
0xc020f458 <yosemite_setup_arch+60>:  lis    r25,-16350
0xc020f45c <yosemite_setup_arch+64>:  bl     0xc000c708
<memcpy>
0xc020f460 <yosemite_setup_arch+68>:  stw    r28,44(r31)
0xc020f464 <yosemite_setup_arch+72>:  li     r4,512
0xc020f468 <yosemite_setup_arch+76>:  li     r5,1
0xc020f46c <yosemite_setup_arch+80>:  li     r3,4116
0xc020f470 <yosemite_setup_arch+84>:  addi   r26,r25,15104
0xc020f474 <yosemite_setup_arch+88>:  bl     0xc000d344
<ocp_get_one_device>
0xc020f478 <yosemite_setup_arch+92>:  lis    r4,-16350
0xc020f47c <yosemite_setup_arch+96>:  lwz    r31,32(r3)
0xc020f480 <yosemite_setup_arch+100>: addi   r4,r4,21534
0xc020f484 <yosemite_setup_arch+104>: li     r5,6
0xc020f488 <yosemite_setup_arch+108>: addi   r3,r31,48
0xc020f48c <yosemite_setup_arch+112>: bl     0xc000c708
<memcpy>
0xc020f490 <yosemite_setup_arch+116>: lis    r4,1017
0xc020f494 <yosemite_setup_arch+120>: lis    r5,168
0xc020f498 <yosemite_setup_arch+124>: stw    r28,44(r31)
0xc020f49c <yosemite_setup_arch+128>: ori    r4,r4,16554
0xc020f4a0 <yosemite_setup_arch+132>: ori    r5,r5,49152
0xc020f4a4 <yosemite_setup_arch+136>: addi   r3,r29,-15380
0xc020f4a8 <yosemite_setup_arch+140>: addi   r29,r29,-15380
0xc020f4ac <yosemite_setup_arch+144>: bl     0xc020e338
<ibm440gx_get_clocks>
0xc020f4b0 <yosemite_setup_arch+148>: li     r0,0
0xc020f4b4 <yosemite_setup_arch+152>: lis    r11,-16352
0xc020f4b8 <yosemite_setup_arch+156>: ori    r0,r0,50000
0xc020f4bc <yosemite_setup_arch+160>: lwz    r10,12(r29)
0xc020f4c0 <yosemite_setup_arch+164>: lis    r9,-16352
0xc020f4c4 <yosemite_setup_arch+168>: stw    r0,8068(r11)
0xc020f4c8 <yosemite_setup_arch+172>: lwz    r0,84(r26)
0xc020f4cc <yosemite_setup_arch+176>: stw    r10,8136(r9)
0xc020f4d0 <yosemite_setup_arch+180>: mtctr  r0
0xc020f4d4 <yosemite_setup_arch+184>: bctrl  r0
0xc020f4d8 <yosemite_setup_arch+188>: li     r5,64
0xc020f4dc <yosemite_setup_arch+192>: mr     r31,r3
0xc020f4e0 <yosemite_setup_arch+196>: lis    r4,-4288
0xc020f4e4 <yosemite_setup_arch+200>: li     r3,0
0xc020f4e8 <yosemite_setup_arch+204>: bl     0xc000c0f8
<ioremap64>

```

```
End of assembler dump.
(gdb)
```

再次重申,要理解此处代码到底发生了什么事,我们并不需要成为PowerPC汇编语言的专家。注意包括PowerPC的bl指令的汇编代码行。bl是PowerPC中用助记符表示的函数调用。符号函数标签是重要的数据点。经过大致的分析后,我们看到下面汇编列表中的几处函数调用。

地 址	函 数
0xc020f438	ocp_get_one_device()
0xc020f45c	memcpy()
0xc020f474	ocp_get_one_device()
0xc020f48c	memcpy()
0xc020f4ac	ibm440gx_get_clocks()

代码清单14-9是文件yosemite.c的部分源代码文件。与在gdbdisassemble输出代码中找到的函数调用代码相结合,可以看到汇编中的几处函数调用bl都出现在yosemite_set_emacdata()函数中。这些函数都是在gdb遇到断点yosemite_setup_arch()后报告的第116行附近。理解这个异常行为的关键所在就是,注意在yosemite_setup_arch()函数中最初的子函数调用。编译器对函数yosemite_set_emacdata()直接产生了内联函数代码,而不是产生函数调用的bl代码,这和我们源代码进行简单检查后所预测的结果一样。这个内联用法使得gdb在遇到断点的时候产生了误匹配。虽然yosemite_set_emacdata()函数没有用关键字inline来定义,但是GCC编译器为了性能的优化仍然将该函数作为内联函数来生成汇编代码。

代码清单14-9 yosemite.c代码片段

```
109 static void __init yosemite_set_emacdata(void)
110 {
111     struct ocp_def *def;
112     struct ocp_func_emac_data *emacdata;
113
114     /* Set mac_addr and phy mode for each EMAC */
115
116     def = ocp_get_one_device(OCP_VENDOR_IBM, OCP_FUNC_EMAC, 0);
117     emacdata = def->additions;
118     memcpy(emacdata->mac_addr, __res.bi_enetaddr, 6);
119     emacdata->phy_mode = PHY_MODE_RMII;
120
121     def = ocp_get_one_device(OCP_VENDOR_IBM, OCP_FUNC_EMAC, 1);
122     emacdata = def->additions;
123     memcpy(emacdata->mac_addr, __res.bi_enet1addr, 6);
124     emacdata->phy_mode = PHY_MODE_RMII;
125 }
126
...
304
305 static void __init
306 yosemite_setup_arch(void)
307 {
```

```

308     yosemite_set_emacdata();
309
310     ibm440gx_get_clocks(&clocks, YOSEMITE_SYSCLK, 6 * 1843200);
311     ocp_sys_info.opb_bus_freq = clocks.opb;
312
313     /* init to some ~sane value until calibrate_delay() runs */
314     loops_per_jiffy = 50000000/HZ;
315
316     /* Setup PCI host bridge */
317     yosemite_setup_hose();
318
319 #ifdef CONFIG_BLK_DEV_INITRD
320     if (initrd_start)
321         ROOT_DEV = Root_RAM0;
322     else
323 #endif
324 #ifdef CONFIG_ROOT_NFS
325         ROOT_DEV = Root_NFS;
326 #else
327         ROOT_DEV = Root_HDA1;
328 #endif
329
330     yosemite_early_serial_map();
331
332     /* Identify the system */
333     printk( "AMCC PowerPC " BOARDNAME " Platform\n" );
334 }
335

```

总结前述的讨论如下。

- 我们首先在yosemite_setup_arch()处设置了断点。
- 当在调试过程遇到断点后，我们发现此时内核运行对应于源代码的第116行，这离我们定义断点的实际位置相差甚远。
- 我们对yosemite_setup_arch()函数进行反汇编分析，发现在反汇编后生成的汇编代码序列中，代码执行流程产生了分支。
- 将汇编代码中的函数调用和源代码比较后，我们发现编译器将函数yosemite_set_emacdata()中设置断点的子函数用内联的方式生成代码，由此引起了潜在的混乱。

这样就解释清楚了，为什么gdb在遇到函数yosemite_setup_arch()中的原始断点后报告的行号和实际的行号不匹配。

编译器使用多种优化算法对代码进行优化。该例子仅仅示例了其中的一个功能：函数内联。每个算法都可能引起调试者（人和机器）的迷惑不清。其中的难点是理解机器层次的代码正在做什么，并且将它转换成开发者打算做的。现在你可以理解使用编译器的最小优化选项为调试所带来的好处了。

14.3.3 gdb 用户定义命令

你可能已经知道gdb在启动阶段会寻找一个初始化文件.gdbinit。当gdb第一次被调用的时

候，gdb加载该初始化文件（通常在用户的home目录），然后会按照该文件里面的命令来响应执行。其中最常用的一个组合是连接到目标系统然后设置初始化断点。在该情形下，.gdbinit文件的内容应该看起来像代码清单14-10。

代码清单14-10 简单的gdb初始化文件

```
$ cat ~/.gdbinit
set history save on
set history filename ~/.gdb_history
set output-radix 16

define connect
#   target remote bdi:2001
   target remote /dev/ttyS0
   b panic
   b sys_sync
end
```

这个简单的.gdbinit文件使gdb能将历史命令存储到一个用户声明的文件中，并且设置打印值的默认输出基数。然后定义了一个用户自定义的命令connect（用户自定义的命令通常称作宏）。当在gdb的提示符下敲入（connect）的时候，gdb会按照指定的方法连接到目标系统并且在panic()和sys_sync()处设置系统断点。该文件中有一个方法可以用来注释代码，我们将在14.4节中讨论该方法。

创造性地使用用户自定义gdb命令是无止境的。用户自定义命令在内核中调试的时候，对于检查全局数据结构（比如进程链表和内存映射等）非常有效。这里我们列出几个实用的gdb用户自定义命令，这些命令可以用来显示内核调试过程中可能需要访问的特定内核数据。

14.3.4 有用的内核 gdb 宏

在内核调试过程中，查看系统中正在运行的进程的信息将对调试非常有用，这些进程都有一些公共的属性。内核中保存了一个由task_struct结构描述进程的链表。链表中第一个进程的地址保存在内核的全局变量init_task中，该变量代表内核在启动阶段产生的初始化进程。每个进程都包括一个list_head链表节点的结构，该结构用来将进程链接成一个环形链表。这两个经常使用的内核结构在下面的头文件中定义：

```
struct task_struct    .../include/linux/sched.h
struct list_head      .../include/linux/list.h
```

使用gdb宏，我们可以遍历进程链表，并且显示进程中有用的信息。可以很方便地修改宏，从而找出你所感兴趣的数据。它是一个非常优秀的学习内核细节的工具。

我们要检查的第一个宏用于查找task_struct结构的进程链表（见代码清单14-11），直到找到给定进程。当找到给定进程后，打印进程的名字。

代码清单14-11 gdb的find_task宏

```
1 # Helper function to find a task given a PID or the
```



```

2 # address of a task_struct.
3 # The result is set into $t
4 define find_task
5   # Addresses greater than _end: kernel data...
6   # ...user passed in an address
7   if ((unsigned)$arg0 > (unsigned)&_end)
8     set $t=(struct task_struct *)$arg0
9   else
10    # User entered a numeric PID
11    # Walk the task list to find it
12    set $t=&init_task
13    if (init_task.pid != (unsigned)$arg0)
14      find_next_task $t
15      while (&init_task!=$t && $t->pid != (unsigned)$arg0)
16        find_next_task $t
17      end
18      if ($t == &init_task)
19        printf "Couldn't find task; using init_task\n"
20      end
21    end
22  end
23  printf "Task \"%s\":\n", $t->comm
24 end

```

将上面的文档内容写到.gdbinit文件中，然后重启gdb，或者使用gdb的source命令将它作为源文件^①。（我们将在代码清单14-15中解释find_next_task宏。）下面是调用find_task的例子：

```

(gdb) find_task 910
Task "syslogd":

```

或

```

(gdb) find_task 0xCFFDE470
Task "bash":

```

第4行定义了宏的名字。第7行判断输入参数是进程号PID（数字被限制从0开始到几百万）还是一个task_struct结构的地址，此地址必须比Linux内核映像地址要大，内核映像地址被符号_end^②所定义。如果它是一个地址，唯一要求采取的行动是将它转化成正确的类型并且用它来解除它对关联task_struct的引用。这是在第8行中完成的。如第3行注释所示，该宏返回一个指向进程task_struct结构的指针。

如果输入参数是一个数字PID，链表将被遍历所有链表以找到匹配的task_struct。第12行和第13行初始化循环变量（在gdb的宏命令语言中没有for循环语句），第15行和第17行定义了寻找的循环体。find_next_task宏用于找到进程链表中下一个task_struct结构的指针。最终，如果查找失败，仍然会设置一个完整的返回值（init_task地址）并且返回，所以它可以在其他

① 一个开发gdb宏的快捷方式是gdb的source命令。该命令打开并读取包含该宏定义的源文件。

② 内核符号_end在内核的最后链接中被定义在链接脚本中。

宏中被安全地使用。

有了代码清单14-11中所定义的find_task宏后，我们可以很容易地创建一条简单的ps命令，来显示系统中运行的每个进程的信息。

代码清单14-12显示了一个gdb宏，用于显示所有进程的实用信息，并且会提取给定进程的task_struct结构。它可以像其他gdb命令一样被调用，输入ps后面再跟上要求的输入参数。注意，这个用户自定义的命令要求一个参数，要么是一个PID要么是task_struct的地址。

代码清单14-12 gdb宏：打印进程信息

```
1 define ps
2   # Print column headers
3   task_struct_header
4   set $t=&init_task
5   task_struct_show $t
6   find_next_task $t
7   # Walk the list
8   while &init_task!=$t
9     # Display useful info about each task
10    task_struct_show $t
11    find_next_task $t
12  end
13 end
14
15 document ps
16 Print points of interest for all tasks
17 end
```

这个ps宏和find_task宏很相似，所不同的是它要求不带参数输入，它增加一个宏(task_struct_show)用于显示每个task_struct的有用信息。第3行带列头打印标识信息行。第4~6行建立循环并且显示第一个进程。第8~11行对每个进程进行循环，对每个进程调用task_struct_show宏来显示每个进程的信息。

注意上面代码中gdb的document命令，接在后面一行是Print points of interest for all tasks，这样允许用户可以在gdb提示符提示下通过输入help ps得到帮助：

```
(gdb) help ps
Print points of interest for all tasks
```

代码清单14-13显示了在一个运行最少服务的目标板上运行ps宏的输出结果。

代码清单14-13 gdb ps宏的输出

```
(gdb) ps
Address      PID State      User_NIP    Kernel-SP    device comm
0xC01D3750   0 Running      0x0FF6E85C  0xC0205E90  (none) swapper
0xC04ACB10   1 Sleeping    0x0FF6E85C  0xC04FFCE0  (none) init
0xC04AC770   2 Sleeping    0x0FF6E85C  0xC0501E90  (none) ksoftirqd/0
0xC04AC3D0   3 Sleeping    0x0FF6E85C  0xC0531E30  (none) events/0
0xC04AC030   4 Sleeping    0x0FF6E85C  0xC0533E30  (none) khelper
0xC04CDB30   5 Sleeping    0x0FF6E85C  0xC0535E30  (none) kthread
0xC04CD790  23 Sleeping    0x0FF6E85C  0xC06FBE30  (none) kblockd/0
```

```

0xC04CD3F0    45 Sleeping      0xC06FDE50 (none) pdflush
0xC04CD050    46 Sleeping      0xC06FFE50 (none) pdflush
0xC054B7B0    48 Sleeping      0xC0703E30 (none) aio/0
0xC054BB50    47 Sleeping      0xC0701E20 (none) kswapd0
0xC054B410   629 Sleeping      0xC0781E60 (none) kseriod
0xC054B070   663 Sleeping      0xCFC59E30 (none) rpciod/0
0xCFFDE0D0   675 Sleeping  0x0FF6E85C 0xCF86DCE0 (none) udevd
0xCF95B110   879 Sleeping  0x0FF0BE58 0xCF517D80 (none) portmap
0xCFC24090   910 Sleeping  0x0FF6E85C 0xCF61BCE0 (none) syslogd
0xCF804490   918 Sleeping  0x0FF66C7C 0xCF65DD70 (none) klogd
0xCFE350B0   948 Sleeping  0x0FF0E85C 0xCF67DCE0 (none) rpc.statd
0xCFFDE810   960 Sleeping  0x0FF6E85C 0xCF5C7CE0 (none) inetd

0xCFC24B70   964 Sleeping  0x0FEEBEAC 0xCF64FD80 (none) mvltd
0xCFE35B90   973 Sleeping  0x0FF66C7C 0xCFEF7CE0 ttyS1  getty
0xCFE357F0   974 Sleeping  0x0FF4B85C 0xCF6EBCE0 (none) in.telnetd
0xCFFDE470   979 Sleeping  0x0FEB6950 0xCF675DB0 ttyp0  bash
0xCFFDEBB0  982<Running  0x0FF6EB6C 0xCF7C3870 ttyp0  sync
(gdb)

```

该ps宏所做的大部分最重要的工作都是task_struct_show宏来完成的。如代码清单14-13所示，task_struct_show宏会显示每个进程的task_struct的一些字段，这些字段在下面列出。

- Address——每一个进程的task_struct的地址；
- PID——进程号；
- State——进程的当前状态；
- User_NIP——用户空间下一条指令指针；
- Kernel_SP——内核栈指针；
- device——和该进程关联的设备；
- comm——进程的名称（或者命令）。

通过修改宏来显示调试过程中你所感兴趣的字段，相对而言是很容易的。其中唯一有难度的地方是宏语言太简单。因为在gdb用户自定义命令语言中没有和strlen等类似的函数，所以屏幕格式化工作必须手工处理。

代码清单14-14再次列出了可以生成前面代码的task_struct_show宏。

代码清单14-14 gdb task_struct_show宏

```

1 define task_struct_show
2   # task_struct addr and PID
3   printf "0x%08X %5d", $arg0, $arg0->pid
4
5   # Place a '<' marker on the current task
6   # if ($arg0 == current)
7   # For PowerPC, register r2 points to the "current" task
8   if ($arg0 == $r2)
9     printf "<"
10  else
11    printf " "
12  end

```

```

13
14 # State
15 if ($arg0->state == 0)
16     printf "Running  "
17 else
18     if ($arg0->state == 1)
19         printf "Sleeping  "
20     else
21         if ($arg0->state == 2)
22             printf "Disksleep "
23         else
24             if ($arg0->state == 4)
25                 printf "Zombie  "
26             else
27                 if ($arg0->state == 8)
28                     printf "sTopped  "
29                 else
30                     if ($arg0->state == 16)
31                         printf "Wpaging  "
32                     else
33                         printf "%2d      ", $arg0->state
34                     end
35                 end
36             end
37         end
38     end
39 end
40
41 # User NIP
42 if ($arg0->thread.regs)
43     printf "0x%08X ", $arg0->thread.regs->nip
44 else
45     printf "      "
46 end
47
48 # Display the kernel stack pointer
49 printf "0x%08X ", $arg0->thread.ksp
50
51 # device
52 if ($arg0->signal->tty)
53     printf "%s  ", $arg0->signal->tty->name
54 else
55     printf "(none) "
56 end
57
58 # comm
59 printf "%s\n", $arg0->comm
60 end

```

第3行显示进程的task_struct地址，第8~12行显示进程PID。如果是当前正在运行的进程（当遇到断点时正在CPU上运行的进程），用<字符标记。

第14~39行解析了进程的状态，并打印出来。紧跟着的是显示用户进程下一条指令指针(NIP)和内核栈指针(SP)。最后，与进程关联的设备会被显示出来，后面跟着进程名（该项存储在task_struct的common字段中）。

一定要注意这个宏是与体系结构相关的，如第7行和第8行代码所示。通常情况下，这样的宏是与体系结构、与版本是紧密相关的。任何时候只要底层结构发生变化，这些宏就必须更新。尽管如此，如果你使用gdb花费大量的时间调试内核，使用宏所带来的好处是对得起你所付出的努力的。

出于完整性的考虑，我们把find_next_task宏列出来。它的实现原理不是很明显，所以有必要做一些解释。（它假定你可以很容易地减少task_struct_header头的内容，以满足本段中出现的ps宏的一系列需求。其实它只不过是用正确数量的空格作为列头的单独一行而已。）代码清单14-15显示了在ps宏和find_task宏中使用的find_next_task宏。

代码清单14-15 gdb find_next_task宏

```
define find_next_task
# Given a task address, find the next task in the linked list
set $t = (struct task_struct *)$arg0
set $offset=(char *)&$t->tasks - (char *)$t
set $t=(struct task_struct *)((char *)$t->tasks.next- (char *)$offset)
end
```

这个宏的功能很简单。它的实现部分也很直接。该宏的目标是返回->next指针，该指针指向进程链表中的下一个进程。但是task_struct结构是被名为tasks的struct list_head的地址成员所链接起来，所以task_struct结构中指向下一个进程的指针不是指向task_struct结构的开始地址。因为->next指针指向进程列表上下一个task_struct中的task结构成员的地址，所以必须将->next地址减去成员的偏移量来获得下一个task_struct的头部地址。我们需要减掉的这个偏移量是指针地址离task_struct头部的距离。首先计算该偏移量，然后使用偏移量来修正->next指针的值，从而获得task_struct的地址。图14-5说明了该过程。

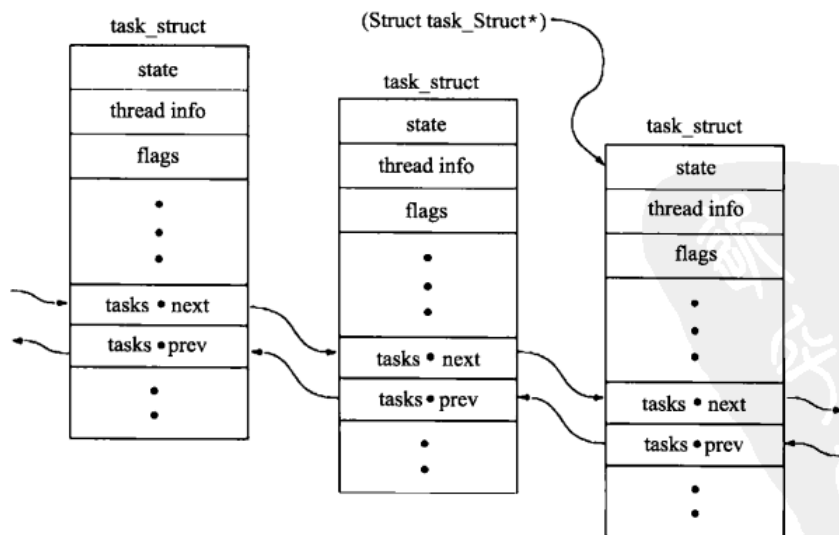


图14-5 进程结构链表

现在我们给出最后一个宏，该宏将在讲解“调试可加载模块”时非常有用。代码清单14-16是一个简单的宏，用来显示当前已经加载的内核模块的信息。

代码清单14-16 gdb列出模块的宏

```

1 define lsmod
2   printf "Address\t\tModule\n"
3   set $m=(struct list_head *)&modules
4   set $done=0
5   while ( !$done )
6     # list_head is 4-bytes into struct module
7     set $mp=(struct module *)((char *)$m->next - (char *)4)
8     printf "0x%08X\t%s\n", $mp, $mp->name
9     if ( $mp->list->next == &modules)
10      set $done=1
11    end
12    set $m=$m->next
13  end
14 end
15
16 document lsmod
17 List the loaded kernel modules and their start addresses
18 end

```

这个简单的循环从内核的全局变量module开始。module这个变量是一个list_head结构，用来标记可加载模块在链表中的起始位置。唯一复杂的地方和代码清单14-15中描述的一样。我们必须从list_head结构指针中减去一个偏移地址来获得结构module的头部地址。这在第7行完成。这个宏产生一个简单的、包含结构module地址和module名字的模块列表。下面是使用lsmod的例子：

```

(gdb) lsmod
Address      Module
0xD1012A80   ip_conntrack_tftp
0xD10105A0   ip_conntrack
0xD102F9A0   loop
(gdb) help lsmod
List the loaded kernel modules and their start addresses
(gdb)

```

这里的宏对于调试都能起到非常大的帮助。你可以用类似的方法创建宏，用来显示内核中的任何内容，使内核更容易访问，尤其是访问内核中由链表维护的主要数据结构。这些宏例子包括进程内存映射信息、模块信息、文件系统信息和定时器信息，等等。这里提供的信息应该可以帮助你入门了。

14.3.5 调试可加载模块

使用KGDB的最常见原因是为了调试可加载的内核模块，也就是调试设备驱动程序。可加载模块最有用的特点之一是，在大多数情形下，没有必要针对每个新的调试会话重启内核。你可以开启一个调试会话，执行一些变化，重新编译内核模块，然后重新加载它们，避免了经常重启内

核的麻烦和延迟。

调试可加载模块的难点是获得模块对象文件中的符号调试信息。因为可加载模块加载到内核时，它们是动态链接的。仅仅包括模块对象文件的符号信息并没有用的，因为符号表需要校正。

回顾前面例子中是如何调用gdb为内核产生调试会话的：

```
$ ppc_4xx-gdb vmlinux
```

这将在主机上开启一个gdb调试会话，并且从Linux的ELF格式内核文件vmlinux中读取符号信息。显然，你在此文件中将找不到任何可加载模块的符号。可加载模块是单独编译单元并且被单独链接成ELF格式文件。因此，如果打算开始任何源代码级的内核模块调试，我们都需要从ELF文件中加载它的调试符号。gdb通过它的add-symbol-file命令提供加载符号信息功能。

add-symbol-file命令从指定模块对象文件中加载符号信息，前提是模块自身已经被加载到内核中。不过调试过程中，我们要面对是“先有鸡还是先有蛋”的哲学问题。我们知道内核模块只有被加载到内核中才有调试符号信息，add-symbol-file命令被用于读取模块的调试符号信息。但是，当模块被加载后，在模块中设置的断点并调试*_init初始化相关函数已经迟了，因为它们已经被执行完。

可以这样来解决这个难题：在内核模块已经被链接但是初始化函数被调用之前，在负责内核模块加载的内核代码中设置断点。这个工作由内核文件.../kernel/module.c完成。代码清单14-17列出module.c文件的部分内容。

代码清单14-17 module.c:模块初始化

```
...
1901     down(&notify_mutex);
1902     notifier_call_chain(&module_notify_list, MODULE_STATE_COMING, mod);
1903     up(&notify_mutex);
1904
1905     /* Start the module */
1906     if (mod->init != NULL)
1907         ret = mod->init();
1908     if (ret < 0) {
1909         /* Init routine failed: abort. Try to protect us from
1910            buggy refcounters. */
1911         mod->state = MODULE_STATE_GOING;
1912     }
1913     ...
```

我们使用modprobe工具加载模块，该命令在代码清单8-5列出，命令用法如下：

```
$ modprobe loop
```

该命令将发布一个特殊的系统调用让内核来加载模块。待加载的模块首先从module.c的sys_init_module()函数处开始运行。当模块被加载到内核内存中并且被动态链接后，控制权交给模块的_init函数。代码清单14-17中第1906行和第1907行显示了这一点。我们在第1907行设置了断点。这就可以确保将内核模块符号信息加载到gdb中，并且随后在模块中设置断点。我们使用Linux内核的回路驱动程序模块loop.ko来演示该过程。loop.ko模块不依赖于其他任何模块，很容易演示。

代码清单14-18显示的gdb命令用于初始化loop.ko的调试会话。

代码清单14-18 初始化模块调试会话: loop.ko

```

1 $ ppc-linux-gdb --silent vmlinux
2 (gdb) connect
3 breakinst () at arch/ppc/kernel/ppc-stub.c:825
4 825      }
5 Breakpoint 1 at 0xc0016b18: file kernel/panic.c, line 74.
6 Breakpoint 2 at 0xc005a8c8: file fs/buffer.c, line 296.
7 (gdb) b module.c:1907
8 Breakpoint 3 at 0xc003430c: file kernel/module.c, line 1907.
9 (gdb) c
10 Continuing.
11 >>> Here we let the kernel finish booting
12     and then load the loop.ko module on the target
13
14 Breakpoint 3, sys_init_module (umod=0x30029000, len=0x2473e,
15     uargs=0x10016338 "") at kernel/module.c:1907
16 1907         ret = mod->init();
17 (gdb) lsmod
18 Address      Module
19 0xd102f9a0    loop
20 (gdb) set $m=(struct module *)0xd102f9a0.
21 (gdb) p $m->module_core
22 $1 = (void *) 0xd102c000
23 (gdb) add-symbol-file ./drivers/block/loop.ko 0xd102c000
24 add symbol table from file "./drivers/block/loop.ko" at
25     .text_addr = 0xd102c000
26 (y or n) y
27 Reading symbols from /home/chris/sandbox/linux-2.6.13-amcc/
    drivers/block      /loop.ko...done.

```

从第2行开始, 我们使用gdb用户自定义的宏connect, 该宏在代码清单14-10中创建, 用于连接到目标板并且设置初始化断点。然后在module.c中增加断点, 如第7行所示, 我们输入继续运行命令c。现在内核完成启动过程, 我们建立一个登录到目标机的telnet会话, 并且将loop.ko模块加载到内核(此过程没有显示出来)。当回环模块被加载后, 我们很快就会遇到第三个断点。gdb于是显示第14~16行的信息。

至此, 我们需要找到Linux内核链接到内核模块.text段的地址。Linux将此地址信息存储在module_core成员元素中的模块信息结构struct module中。使用代码清单14-16中定义的lsmod宏, 我们获得与loop.ko模块关联的struct module的地址。第17~19行显示此过程。现在我们将使用该结构地址从module_core结构成员中获取模块的.text段地址。我们将该地址传递给gdb的add-symbol-file命令, 于是gdb使用该地址去校正它的内部符号表以匹配模块被链接进内核的真实地址。顺利完成之后, 我们就可以像普通调试一样在模块中设置断点、跟踪代码、检查数据等。

本节中我们将演示, 在回环模块的初始化函数中设置断点, 就可以跟踪回环模块的初始化代码。在这里要注意的是, 内核将模块的初始化代码加载到一个单独分配的内存区域, 所以当使用

完后可以被释放。回顾第5章所讲的内容，我们讨论过__init宏。该宏扩展编译器的属性，利用该属性可以指导链接器将标记的代码部分放置到ELF文件中。实际上，任何函数使用该属性定义后，都将放到一个单独的名为.init.text的ELF段中。它的用法和下面的相似：

```
static int __init loop_init(void){...}
```

这个__init符号将把函数loop_init()编译到loop.ko对象模块的.init.text节内容中。当模块被加载的时候，内核为模块的主体部分分配一大块内存，这块内存区域被结构模块成员module_core所指向。然后内核将为.init.text段分配一个单独的内存区域。在调用初始化函数之后，内核会释放包含初始化函数的内存。因为模块对象是这样被分开存放的，我们需要告诉gdb这种寻址机制，来让gdb使用符号数据调试初始化函数^①。代码清单14-19展现了这些步骤。

代码清单14-19 用来调试init模块的代码

```
$ ppc_4xx-gdb -slient vmlinux
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
breakinst () at arch/ppc/kernel/ppc-stub.c:825
825      }
<< Place a breakpoint before calling module init >>
(gdb) b module.c:1907
Breakpoint 1 at 0xc0036418: file kernel/module.c, line 1907.
(gdb) c
Continuing.

Breakpoint 1, sys_init_module (umod=0xd102ef40, len=0x23cb3, uargs=0x10016338 "")
at kernel/module.c:1907
1907      ret = mod->init();

<< Discover init addressing from struct module >>
(gdb) lsmod
Address      Module
0xd102ef40   loop
(gdb) set $m=(struct module *)0xd102ef40
(gdb) p $m->module_core
$1 = (void *) 0xd102b000
(gdb) p $m->module_init
$2 = (void *) 0xd1031000
<< Now load a symbol file using the core and init addrs >>
(gdb) add-symbol-file ./drivers/block/loop.ko 0xd102b000 -s .init.text 0xd1031000
add symbol table from file "./drivers/block/loop.ko" at
      .text_addr = 0xd102b000
      .init.text_addr = 0xd1031000
(y or n) y
Reading symbols from /home/chris/sandbox/linux-2.6.13-
amcc/drivers/block/loop.ko...done.
(gdb) b loop_init
Breakpoint 3 at 0xd1031000: file drivers/block/loop.c, line 1244.
```

① 创作本书时，gdb中有一个很大的bug导致这个技巧不能正常工作。希望下次你再读到这里的时候，这个问题已经得到修复。

```
(gdb) c
Continuing.
<< Breakpoint hit, proceed to debug module init function >>
Breakpoint 3, 0xd1031000 in loop_init () file drivers/block/loop.c, line 1244
1244         if (max_loop < 1 || max_loop > 256) {
(gdb)
```

14.3.6 printk 调试

printk是调试内核和设备驱动程序的一种常用的技术，主要是因为printk已经发展成了一个非常稳定的方法。你可以在几乎任何地方调用printk，包括在中断处理里面。printk是内核版本的C库函数printf()。printk在.../kernel/printk.c中定义。

掌握printk的使用限制对于调试而言非常重要。首先printk需要一个控制台设备。而且该控制台设备在内核初始化过程中要尽可能早地配置，在控制台设备被初始化之前有很多调用都要用到printk。我们在14.5节将介绍一种方法来处理该限制。

printk函数允许增加一个字符串来标记一个给定消息的安全级别。头文件.../include/linux/kernel.h定义了8个级别：

```
#define KERN_EMERG    "<0>" /* system is unusable */
#define KERN_ALERT    "<1>" /* action must be taken immediately */
#define KERN_CRIT     "<2>" /* critical conditions */
#define KERN_ERR      "<3>" /* error conditions */
#define KERN_WARNING   "<4>" /* warning conditions */
#define KERN_NOTICE    "<5>" /* normal but significant condition */
#define KERN_INFO      "<6>" /* informational */
#define KERN_DEBUG     "<7>" /* debug-level messages */
```

简单的printk消息看起来如下所示：

```
printk("foo() entered w/ %s\n", arg);
```

如果定义安全级别的字符串被忽略，那么内核将分配一个默认的安全级别，默认值在printk.c中定义。在目前的内核中，该级别被设置成第4级——KERN_WARNING。定义安全级别的printk示例如下：

```
printk(KERN_CRIT "vmalloc failed in foo()\n");
```

默认情况下，所有预先定义日志级别的printk消息都会显示在系统的控制台设备上。默认的日志级别在printk.c文件中定义。在新版Linux内核中，一般会将该日志级别设定为7。这就意味着任何比KERN_DEBUG重要的printk消息都将在控制台显示出来。

你可以有很多方法来设置默认的内核日志级别。可以在内核启动时刻传递给内核确切的内核命令行参数，在内核参数中定义目标板的默认日志级别。main.c文件定义了3个内核命令行选项，它们可以影响默认的日志级别：

- debug——把控制台日志级别设为10；
- quiet——把控制台日志级别设为4；
- loglevel——根据自选值设定控制台日志级别。

使用debug调试模式可以有效地显示任何一条printk消息。使用quiet安静模式显示KERN_ERR级别以及更高级别的printk消息。

printk消息可以把日志记录到目标系统文件中，或通过网络进行记录。使用klogd（内核日志守护进程）和syslogd（系统守护进程）可以控制printk消息的记录日志的行为。这些常用的工具在man帮助手册和大量的Linux参考中都有讲述，这里将不再介绍。

14.3.7 Magic SysReq 键

该调试手段会首先定义好一系列预定义的键序列，然后直接向内核发送消息。对于很多目标体系结构和开发板，可以使用在串口上的简单的终端模拟器作为系统的控制台。对于这些体系结构而言，Magic SysReq键被定义成一个中断符，后面紧跟着一个命令字符。关于如何发送一个中断符可以查阅终端模拟器使用文档。有很多Linux开发者使用minicom终端模拟器。对于minicom而言，中断符是靠敲入Ctl-A F来发送的。在用这种方式发送了中断后，你有5秒的时间在命令超时之前输入命令字符。

这个内核工具对开发和调试起到非常大的帮助。但是它也可能会引起数据丢失和系统破坏。其实，b命令可以不带任何提示或者准备来立即重启系统。打开的文件不能被关闭，磁盘不能同步，文件系统没有被卸载。当输入重启命令b后，控制权会以生硬的方式立即移交给体系结构的复位向量。使用这个强大的工具会带有很大的危险性！

该功能在Linux内核文档的一个子目录sysrq.txt中很好地进行了描述。在其中你可以找到很多体系结构的细节以及有效命令的描述。

例如，另一个设置内核日志级别的方法就是使用Magic SysReq键。该命令是0~9中的一个数字，它将导致默认的日志级别被设置成该命令的数字。对于minicom，按Ctl-A F后再输入一个数字，比如9。下面是该命令终端显示的例子：

```
$ SysRq : Changing Loglevel
Loglevel set to 9
```

该命令还可以用于查看寄存器、关闭系统、重启系统、查看进程链表、查看当前控制台的内存信息等。可以在任何Linux内核文档里查看详细内容。

该功能通常用于由于某些原因而引起系统死机的时候。Magic SysReq键通常会提供一种方法可以访问挂起的系统。

14.4 硬件辅助调试

至此，你已经了解不能用KGDB调试Linux内核的早期启动代码，这是因为KGDB在大多数更底层的硬件初始化代码被执行后才被初始化。而且，如果你被指派要完成一个全新开发版的设计工作，而且要移植引导装入程序和Linux内核，硬件调试探测器无疑将是最有效的手段，可以用来解决板探测的早期阶段中的调试问题。

你可以从大量的硬件调试探测器中进行选择。在本节的例子中，我们使用一个由Abatron生产的调试器：BDI-2000（www.abatron.ch）。这些类型的调试器通常被称为JTAG探测器，因为它

们使用一种底层的通信方式，即最初是由联合测试行动小组（JTAG）提出的集成电路边界扫描测试技术。

JTAG探测器包含一个小型的连接器，利用它连接到目标板。连接器的外观通常是一个方形的头，后面是带状的电缆。现代大部分高性能的CPU都会提供一个JTAG的接口，此接口被设计用于这种软件调试。JTAG探测器连接到目标机CPU的JTAG接口上。同时，JTAG探测器通过以太网、USB或者并口连接主机开发系统。图14-6详细地显示了Abatron调试器的建立过程。

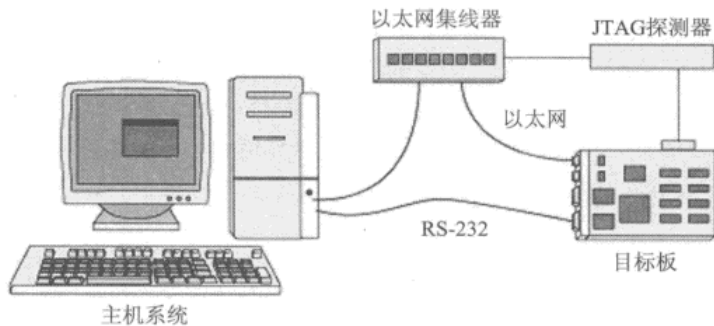


图14-6 硬件JTAG探测器连接

JTAG探测器的建立过程比较复杂，这是将CPU连接到JTAG的复杂性的直接结果。当目标板加电之后，CPU重新复位，此时硬件都没有初始化。事实上，很多处理器在正式处理事务之前需要做一些初始化工作。有很多方法可以用于将初始化配置信息设置到CPU中。一些CPU会读取硬件配置字或者特定管脚的初始化值来获知加电时的配置。其他的CPU读取非失易存储设备（比如闪存）的默认位置来初始化。当使用一个JTAG探测器，尤其是用于启动一个新板子的设计时，在做任何事情之前必须完成一个最低层次的CPU和板的初始化工作。很多JTAG探测器依赖于这个初始化的配置文件。

Abatron调试器使用一个配置文件去初始化它所连接的目标硬件，以及在配置文件中定义调试时的其他可操作参数。这个配置文件包括用于初始化CPU、内存和其他必需的板级硬件的指令。开发者的职责就是用正确的针对板的指令来自定义配置文件。这些配置文件的命令细节可以在JTAG探测器文档中找到。但是，只有嵌入式开发者才能创建特定板设计所需的唯一的配置文件。这要求对CPU和对板级设计特点有详细的了解。与为新板创建自定义Linux端口一样，没有捷径或者其他的替换方式。

附录E包含一个Abatron配置文件示例，针对以Freescale半导体公司生产的MPC5200为控制器的自定义开发板。在这个附录中，你可以看到一个自定义板的必要设置。注意那些用于描述各种寄存器和初始化细节的注释。这将使以后的升级和维护工作变得容易，并且可以帮助你第一次就将配置文件修改正确。

硬件探测器通常有两种使用方法。大部分探测器提供某种类型的用户接口，以使开发者使用探测器的特点。这种用法的示例就是对闪存进行编程，或下载二进制映像文件。第二个用法是作

为gdb或者其他源代码调试器的前端。我们将分别展示这两种使用技巧。

14.4.1 使用 JTAG 探测器对闪存编程

很多硬件探测器都包括了对不同系列闪存芯片进行编程的能力。Abatron BDI-2000也不例外。BDI-2000的配置文件包括一个用来定义目标闪存特征的[FLASH]段。请参考附录E中的示例。[FLASH]段定义了用于特定设计中的闪存芯片的属性，比如芯片类型、设备的大小、数据总线宽度。同时定义了在内存中的位置和描述芯片存储组织的一些方式。

当升级闪存的一部分的时候，你经常需要保留相同闪存的其他内容。在这种情形下，硬件探测器必须有某种方法限制被擦除的段。在Abatron系列中，这个问题是这样做到的：增加以ERASE关键字开头的一行，ERASE要加在某个节前面表示该节需要擦除。通过telnet登录到Abatron后，使用erase命令擦除的时候，所有被定义成ERASE段的信息都会被擦除掉。代码清单14-20演示了如何擦除目标板上闪存的部分内容，以及随后将U-Boot引导装入程序映像编程到闪存的过程。

代码清单14-20 擦除和对闪存编程

```
$ telnet bdi
Trying 192.168.1.129...
Connected to bdi (192.168.1.129).
Escape character is '^]'.
BDI Debugger for Embedded PowerPC
=====
... (large volume of help text)

uei> erase
Erasing flash at 0xffff00000
Erasing flash at 0xffff10000
Erasing flash at 0xffff20000
Erasing flash at 0xffff30000
Erasing flash at 0xffff40000
Erasing flash passed
uei> prog 0xffff00000 u-boot.bin BIN
Programming u-boot.bin , please wait ...
Programming flash passed
uei>
```

首先我们建立一次telnet会话连接到Abatron BDI-2000。经过一些初始化之后，出现一个命令提示符。使用erase命令后，Abatron显示配置文件中定义的每个段的输出信息。如同附录E所示的配置信息，我们定义了5个要擦除的段。这些段加起来的大小有256KB，用于保存U-Boot的二进制映像。

prog命令列出它的3个可选参数。这些参数定义了新映像被加载到内存中的位置和映像的名字、映像的格式（BIN，一个二进制文件）。你可以在BDI-2000的配置文件中定义这些参数，这样prog命令就可以变成不带任何参数来使用。

该例子仅仅提到BDI-2000的两个命令。BDI-2000还支持更多复合的命令和强大的功能。每一个硬件JTAG探测器都有它自己的方式擦写和编程闪存。可以查阅特定设备的规范文档。

14.4.2 用 JTAG 探测器进行调试

很多JTAG探测器不采用直接通过JTAG用户界面进行交互，而是和源代码探测器接口交互。目前大多数硬件探测器支持的源代码探测器是gdb。在这个使用情景下，gdb通过外部连接，通常采用以太网连接到目标板，然后开始一个调试会话。不是直接采用JTAG探测器用户界面进行通信，而是让gdb调试器在自身和JTAG探测器之间来回传送命令。此种模型下，JTAG探测器使用gdb代表调试器，使用远程协议去控制硬件。回顾图14-6，阅读连接细节。

JTAG探测器对引导装入程序和早期启动代码的源代码级调试非常有效。本例中，我们将说明如何用gdb和Abatron BDI-2000来调试一个PowerPC目标板上U-Boot的部分代码。

很多处理器都包含一些调试寄存器，这些寄存器具有设置传统的地址断点（当程序运行到某个地址的时候停下）、设置数据断点（当访问到指定内存地址满足某条件时停下）的功能。当调试存放在只读内存（例如闪存）中的代码时，这是设置断点的唯一方法。但是，这些寄存器的数目通常会很有限。很多处理器只包括一个或者两个这样的寄存器。在使用硬件探测器设置断点之前，必须理解这个限制。下面的例子可以说明这一点。

使用图14-6所示的配置建立一次连接，假设目标板已经在闪存中存有U-Boot。在第7章中介绍了，U-Boot和其他引导装入程序在启动之后会尽可能早地将自身复制到RAM中。这是因为硬件从RAM中的读（写）周期要比从只读存储设备（例如闪存）中的读（写）周期快好几个数量级。因此带来了两个调试挑战。首先，我们不能修改只读内存中的内容（比如插入一个软件断点），必须依赖于处理器支持的断点寄存器达到这个目的。

第二个挑战源于闪存或者RAM中唯一的执行上下文，gdb从ELF可执行文件中读取符号调试信息的时候只有唯一的执行上下文。在U-Boot的例子中，它针对存储在闪存中初始化位置而进行链接。早期启动阶段，代码重定位自己并且要做很多必需的位置调整。这表明我们需要在这两种执行上下文中来进行gdb调试工作。代码清单14-21显示了这样调试会话的例子。

代码清单14-21 使用JTAG探测器调试U-Boot

```
$ ppc-linux-gdb --silent u-boot
(gdb) target remote bdi:2001
Remote debugging using bdi:2001
_start () at /home/chris/sandbox/u-boot-1.1.4/cpu/mpc5xxx/start.S:91
91      li      r21, BOOTFLAG_COLD /* Normal Power-On */
Current language: auto; currently asm

<< Debug a flash resident code snippet >>
(gdb) mon break hard
(gdb) b board_init_f
Breakpoint 1 at 0xffff0457c: file board.c, line 366.
(gdb) c
Continuing.

Breakpoint 1, board_init_f (bootflag=0x7fc3afc) at board.c:366
366      gd = (gd_t *) (CFG_INIT_RAM_ADDR + CFG_GBL_DATA_OFFSET);
Current language: auto; currently c
```

```

(gdb) bt
#0 board_init_f (bootflag=0x1) at board.c:366
#1 0xffff0456c in board_init_f (bootflag=0x1) at board.c:353
(gdb) i frame
Stack level 0, frame at 0xf000bf50:
pc = 0xffff0457c in board_init_f (board.c:366); saved pc 0xffff0456c
called by frame at 0xf000bf78
source language c.
Arglist at 0xf000bf50, args: bootflag=0x1
Locals at 0xf000bf50, Previous frame's sp is 0x0

<< Now debug a memory resident code snippet after relocation >>
(gdb) del 1
(gdb) symbol-file
Discard symbol table from '/home/chris/sandbox/u-boot-1.1.4-powerdna/u-boot'?
(y or n) y
No symbol file now.
(gdb) add-symbol-file u-boot 0x7fa8000
add symbol table from file "u-boot" at
      .text_addr = 0x7fa8000
(y or n) y
Reading symbols from u-boot...done.
(gdb) b board_init_r
Breakpoint 2 at 0x7fac6c0: file board.c, line 608.
(gdb) c
Continuing.
Breakpoint 2, board_init_r (id=0x7f85f84, dest_addr=0x7f85f84) at board.c:608
608          gd = id;      /* initialize RAM version of global data */
(gdb) i frame
Stack level 0, frame at 0x7f85f38:
pc = 0x7fac6c0 in board_init_r (board.c:608); saved pc 0x7fac6b0
called by frame at 0x7f85f68
source language c.
Arglist at 0x7f85f38, args: id=0x7f85f84, dest_addr=0x7f85f84
Locals at 0x7f85f38, Previous frame's sp is 0x0
(gdb) mon break soft
(gdb)

```

仔细研究该例子，有很多精妙的细节值得花时间去理解。首先，我们通过target remote命令连接到Abatron BDI-2000上。本例中的IP地址是Abatron探测器的IP地址，用符号名bdi表示^①。Abatron BDI-2000使用端口2001为远程gdb连接使用。

接下来使用gdb mon命令操作BDI-2000。mon命令告诉gdb将其余命令直接传送给远程目标硬件设备。因此，mon break hard命令将设置BDI-2000进入硬件断点模式。

然后在board_init_f函数设置硬件断点。board_init_f例程是当运行越出闪存地址0xffff0457c处时要执行的程序。断点定义后，输入继续运行命令c恢复运行。很快就遇到在board_init_f处的断点，之后我们可以自由使用常用的调试活动，包括跟踪代码和检查数据。可以看到，我们使用bt命令检查栈的回溯调用，i frame命令检查当前栈帧的详细内容。

① 在主机系统中的/etc/hosts文件里的每一个条目可以使用一个IP地址。

现在继续执行，这次我们知道U-Boot把它自身复制到RAM中，然后从复制到RAM中的副本继续执行。所以我们需要在保持调试会话状态的同时改变调试上下文。为此，我们丢弃当前符号表（用不带参数的symbol-file命令），并且用add-symbol-file命令将同一个符号文件加载进来。此次我们让gdb将符号表位置偏移，使其与U-Boot重定位自身在内存中的位置相匹配。这样确保源代码以及符号调试信息和实际的内存中放置的映像相匹配。

新的符号表加载进来之后，在代码某处增加一个硬件断点，因为我们知道执行的时候该断点处将进驻RAM。这是调试中精妙复杂的一处。因为我们知道U-Boot当前运行在闪存中，但是它正打算将自己复制到RAM中然后跳转到RAM中的副本处。我们必须仍然使用硬件断点。想想如果在此使用一个软件调试断点会发生什么情况，gdb会忠实地将该断点操作码写入指定的内存位置，但是U-Boot将自身复制到RAM时将覆盖它。最终的结果将是永远也不会遇到该断点，于是我们开始怀疑工具损坏了。在U-Boot进入到RAM中的副本，并且符号表信息已经和基于RAM地址的复制同步之后，我们就可以自由使用基于RAM的断点了。如代码清单14-21中的最后一条命令，就是将Abatron探测器的断点模式重新设置成软件断点模式。

为什么我们要关心硬件断点和软件断点呢？如果硬件断点寄存器的个数没有限制，那么我们不用关心这点。但是事实并非如此。下面就是一个在调试会话期间采用硬件断点调试模式时超过处理器支持的断点寄存器个数的例子。

```
(gdb) b Flash_init
Breakpoint 3 at 0x7fbeb0: file Flash.c, line 70.
(gdb) c
Continuing.
warning: Cannot insert breakpoint 3:
Error accessing memory address 0x7fbeb0: Unknown error 4294967295.
```

因为在使用远程调试，只有当增加新的断点并且用continue继续时，我们才知道硬件资源的限制。这是因为gdb处理断点的方式。当遇到一个断点之后，gdb会用指示内存地址的原始操作码恢复所有的断点信息。当它继续执行，它就恢复指定位置处的断点操作码。你可以打开gdb的远程调试模式来观察该行为：

```
(gdb) set debug remote 1
```

14.5 无法启动时

在各种嵌入式Linux的邮件列表中，最常问到的一个问题和下面描述的相似：

我的工作是启动开发板上的Linux，但是在控制台终端上输出这样一条打印消息时就卡住了：
Uncompressing Kernel Image ... OK.

于是我们就开始了嵌入式Linux的一个长时间，甚至有时让你心灰意冷的反复学习过程。启动时很多方面都可能出错，最后导致这个常见的错误信息。只要有JTAG探测器和一些背景知识，我们就能使用几种办法来定位问题的位置。

14.5.1 早期串口调试输出

可以采用的第一个有效工具是CONFIG_SERIAL_TEXT_DEBUG。这个Linux内核配置选项将在启动过程的早期阶段增加对打印调试信息的支持。目前，这个功能只限制在PowerPC体系结构上。代码清单14-22是使用该功能的PowerPC目标板上的一个例子，使用的引导装入程序是U-Boot。

代码清单14-22 早期串口文本调试

```
## Booting image at 00200000 ...
Image Name:   Linux-2.6.14
Created:      2005-12-19  22:24:03 UTC
Image Type:   PowerPC Linux Kernel Image (gzip compressed)
Data Size:    607149 Bytes = 592.9 kB
Load Address: 00000000
Entry Point:  00000000
Verifying Checksum ... OK
Uncompressing Kernel Image ... OK
id mach(): done      <== Start of messages enabled by
MMU:enter            <== CONFIG_SERIAL_TEXT_DEBUG
MMU:hw init
MMU:mapin
MMU:setio
MMU:exit
setup_arch: enter
setup_arch: bootmem
arch: exit
arch: real exit
```

应用该功能，可以判断目标板在启动过程中在哪个地方卡住了。当然你也可以在内核的其他地方增加自己的早期调试信息。下面是一个该用法的例子，修改的文件是.../arch/ppc/mm/init.c:

```
/* Map in all of RAM starting at KERNELBASE */
if (ppc_md.progress)
    ppc_md.progress("MMU:mapin", 0x301);
mapin_ram();
```

AMCC Yosemite平台是该基础设施的非常好的示例。查阅下述Linux源码树^①中的文件，找到调试系统是如何被实现的细节。

文 件	函 数	目 的
gen550_dbg.c	gen550_init	yosemite.c平台初始化文件调用的串口设置
gen550_dbg.c	gen550_progress	底层串口输出例程
ibm44x_common.c	ibm44x_platform_init	将特定平台的例程绑定到通用PPC样机基础设施

① 所有这些文件是唯一的，所以没有完整路径名也可以找到。

14.5.2 转储 printk 日志缓冲区

在14.3.6节讨论printk调试的时候，我们指出一些使用这种方法的限制。printk本身是一个非常健壮的功能实现。它的一个缺点是直到终端设备被初始化后才能看到printk的信息，而终端设备被初始化已经是启动后很靠后的过程了。通常情况下，当开发板卡在启动过程中时，许多信息都会被阻塞在printk的缓冲区中。如果知道到哪儿去找到它们，就可以准确地找到是哪个问题挂起启动过程的。事实上，很多次你将会发现内核启动时遇到一个错误，该错误导致调用panic()。从panic()打印的输出很有可能被放在printk的缓冲区，因此就可以定位到导致出错的代码的准确行号。

这种问题最好在JTAG探测器的帮助下完成，不过也有可能使用引导装入程序和它的内存转储功能，在重新置位后来显示printk缓冲区的内容。有时内存内容的破坏可能导致系统重启，但是日志缓冲中文本信息通常是可以读懂的。

printk存储它的消息文本的缓冲区的位置在printk的源文件.../kernel/printk.c中声明：

```
static char __log_buf[__LOG_BUF_LEN];
```

我们可以很容易地在Linux内核映射文件System.map中找到该缓冲区的链接位置。

```
$ grep __log_buf System.map
c022e5a4 b __log_buf
```

如果系统碰巧在启动过程中，显示了“Uncompressing Kernel Image ... OK”消息后挂起，你可以重启系统，使用引导装入程序去检查缓冲区内容。因为在一个给定的体系结构中，内核虚拟内存和物理内存的关系是固定的，是一个常数，所以我们可以做简单的转换。前面显示的__log_buf的地址是一个内核虚拟地址，我们必须将它转换成物理地址。在这个特定的PowerPC体系结构中，这种转换是一个简单的减法，用虚拟地址减去常量KERNELBASE，0xc0000000。这就是我们在内存中读取内容的地址，如果有内容，printk日志缓冲区都放置在这里。

代码清单14-23是使用U-Boot的内存转储命令显示的内容。

代码清单14-23 转储printk日志缓冲

```
=> md 22e5a4
0022e5a4: 3c353e4c 696e7578 20766572 73696f6e    <5>Linux version
0022e5b4: 20322e36 2e313320 28636872 6973406a    2.6.13 (chris@
0022e5c4: 756e696f 72292028 67636320 76657273    junior) (gcc
0022e5d4: 696f6e20 332e342e 3320284d 6f6e7461    version 3.4.3 (Monta
0022e5e4: 56697374 6120332e 342e332d 32352e30    Vista 3.4.3-25.0
0022e5f4: 2e37302e 30353031 39363120 32303035    .70.0501961 2005
0022e604: 2d31322d 31382929 20233131 20547565    -12-18)) #11 Tue
0022e614: 20466562 20313420 32313a30 353a3036    Feb 14 21:05:06
0022e624: 20455354 20323030 360a3c34 3e414d43    EST 2006.<4>AMC
0022e634: 4320506f 77657250 43203434 30455020    C PowerPC 440EP
0022e644: 596f7365 6d697465 20506c61 74666f72    Yosemite Platform.
0022e654: 6d0a3c37 3e4f6e20 6e6f6465 20302074    <7>On node 0
0022e664: 6f74616c 70616765 733a2036 35353336    totalpages: 65536
```

```

0022e674: 0a3c373e 2020444d 41207a6f 6e653a20 .<7> DMA zone:
0022e684: 36353533 36207061 6765732c 204c4946 65536 pages, LIF
0022e694: 4f206261 7463683a 33310a3c 373e2020 0 batch:31.<7>
=>
0022e6a4: 4e6f726d 616c207a 6f6e653a 20302070 Normal zone: 0
0022e6b4: 61676573 2c204c49 464f2062 61746368 pages, LIFO batch
0022e6c4: 3a310a3c 373e2020 48696768 4d656d20 :1.<7> HighMemzone:
0022e6d4: 7a6f6e65 3a203020 70616765 732c204c 0 pages,
0022e6e4: 49464f20 62617463 683a310a 3c343e42 LIFO batch:1.<4>
0022e6f4: 75696c74 2031207a 6f6e656c 69737473 Built 1 zonelists
0022e704: 0a3c353e 4b65726e 656c2063 6f6d6d61 .<5>Kernel command
0022e714: 6e64206c 696e653a 20636f6e 736f6c65 line: console
0022e724: 3d747479 53302c31 31353230 3020726f =ttyS0,115200
0022e734: 6f743d2f 6465762f 6e667320 72772069 root=/dev/nfs rw
0022e744: 703d6468 63700a3c 343e5049 44206861 ip=dhcp.<4>PID
0022e754: 73682074 61626c65 20656e74 72696573 hash table entries
0022e764: 3a203230 34382028 6f726465 723a2031 : 2048 (order:
0022e774: 312c2033 32373638 20627974 6573290a 11, 32768 bytes).
0022e784: 00000000 00000000 00000000 00000000 .....
0022e794: 00000000 00000000 00000000 00000000 .....
=>

```

代码清单中的内容不是很好读，但是需要的数据都在。我们在该例中可以看到，内核在初始化PID散列表条目后崩溃。有了这些附加的打印信息，我们就可以开始圈定引起系统崩溃的源代码。

就如本例中所示的，这是一个在没有其他硬件工具可用的情况下的技巧。如果你正在做一个新板子的移植工作，你将会发现早期串口输出的重要性。

14.5.3 KGDB 捕捉崩溃

如果KGDB被开启，内核遇到错误时会试图将控制回传给KGDB。某些情形下，错误本身看起来就非常明显。为了使用该功能，在KGDB和gdb之间必须已经建立好连接。当遇到异常情形时，KGDB发出一个Stop Reply的数据包给gdb，指出陷入到调试处理函数的原因，同时也提供陷阱条件发生时的地址。代码清单14-24说明该序列过程。

代码清单14-24 使用KGDB捕捉崩溃

```

$ ppc_4xx-gdb --silent vmlinux
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
Malformed response to offset query, qOffsets
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
breakinst () at arch/ppc/kernel/ppc-stub.c:825
825      }
(gdb) c
Continuing.
<< KGDB gains control from panic() on crash >>
Program received signal SIGSEGV, Segmentation fault.
0xc0215d6c in pcibios_init () at arch/ppc/kernel/pci.c:1263

```

```

1263          *(int *)-1 = 0;
(gdb) bt
#0  0xc0215d6c in pcibios_init () at arch/ppc/kernel/pci.c:1263
#1  0xc020e728 in do_initcalls () at init/main.c:563
#2  0xc020e7c4 in do_basic_setup () at init/main.c:605
#3  0xc0001374 in init (unused=0x20) at init/main.c:677
#4  0xc00049d0 in kernel_thread ()
Previous frame inner to this frame (corrupt stack?)
(gdb)

```

该例子中的崩溃是由简单地往一个无效内存地址中写入而引起的。我们首先建立一个gdb到KGDB的连接，然后允许内核继续启动。要留意我们甚至不用设置断点，当崩溃发生的时候，我们可以获得导致错误的代码的行号，并且得到一个回溯调用序列帮助我们确定错误原因。

14.6 小结

- Linux内核调试非常复杂，尤其在交叉开发环境中。理解这些复杂关系是成功调试内核的关键。
- KGDB是一个非常有用的内核级别的gdb桩，可以开启直接在内核和设备驱动程序中的源代码级别的调试。它使用gdb远程协议和主机上的交叉gdb进行通信。
- 理解（同时最小化）编译器优化选项有助于理解编译器优化代码后与预期不同的行为。
- gdb支持用户自定义命令，这些命令对于自动调试枯燥的任务非常有用，比如在调试重复的内核链表和访问复杂的变量时。
- 内核可加载模块对于源代码级的调试有自身的挑战。模块的初始化例程可以在module.c中module->init()调用处设置断点，
- printk和Magic SysReq键提供了附加工具，可以在内核开发和调试过程中将遇到的问题分离出来。
- 通过JTAG探测器的硬件辅助调试工具可以调试驻在闪存和ROM中的代码，而其他调试方法要么很笨重要么不可能对此调试。
- 打开体系结构中的CONFIG_SERIAL_TEXT_DEBUG功能，在移植新内核的时候，它会成为一种强大的工具。
- 检查printk的日志缓冲区内容通常可以找到引起内核启动时候引起系统崩溃的原因。
- KGDB在内核崩溃后将控制权传递给gdb，可以通过检查回溯调用来找到引起内核崩溃的原因。

参考资料

Linux Kernel Development, 2nd Edition

Robert Love

Novell Press, 2005

The Linux Kernel Primer

Claudia Salzberg Rodriguez et al.
Prentice Hall, 2005

“使用GNU编译器”

Richard M. Stallman and the GCC Developer Community
GNU Press, a division of Free Software Foundation
<http://gcc.gnu.org/onlinedocs/>

KGDB在Sourceforge中的主页
<http://sourceforge.net/projects/KGDB>

使用GDB调试

Richard Stallman, Roland Pesch, Stan Shebs, et al.
Free Software Foundation
www.gnu.org/software/gdb/documentation/

工具接口标准

DWARF 调试信息格式规范, 2.0版
TIS Committee, May 1995



本章内容

- 目标机调试
- 远程（交叉）调试
- 使用共享库进行调试
- 多任务调试
- 远程调试的附加选项
- 小结

在前一章中，我们讲述了GDB在调试内核代码和固化在Flash中的代码（如引导装入程序代码）的使用。在本章中，我们将介绍GDB调试用户空间的应用程序的方法，并将研究范围扩展到嵌入式的交叉开发环境，讲解在这种特殊调试环境下使用的远程调试工具以及技巧。

15.1 目标机调试

我们已经在第13章中介绍了几种重要的调试工具。strace和ltrace可以用来记录和分析进程的行为，从而不断分析出问题所在。dmalloc能够定位内存泄漏问题，剖析内存使用情况。ps和top都是检查处理器运行状态的得力工具。这些相对较小的工具都设计为可以在目标机上直接运行。

在嵌入式系统上调试Linux应用程序面临其自身独有的挑战。嵌入式目标板上的资源通常是受限的。例如RAM和非易失存储设备的限制可能会妨碍你运行基于目标板上的开发工具。目标机可能没有以太网接口或其他高速连接设备。你的嵌入式目标板也可能没有显示设备，没有键盘或鼠标。

使用交叉开发工具和挂载NFS根文件系统就能在调试过程中获得巨大好处。很多工具已经设计为在开发主机上执行，但实际上调试远程目标机上的代码，特别是GDB。GDB不仅可以用于调试目标机上的应用程序，而且还能够分析应用程序崩溃后产生的核心文件。我们在第13章介绍了应用程序核心转储（core dump）的细节内容。

15.2 远程（交叉）调试

设计交叉开发工具的首要目的就是为了克服嵌入式平台资源的限制。一个加上符号调试信息

的普通应用程序很轻易会超过几兆字节大小。使用交叉调试，可以在开发主机上处理这个负担。当你在开发主机上调用交叉版本的GDB时，需要传递给它一个编译有符号调试信息的ELF文件。在目标板那端，你没有理由不除去^①ELF文件中所有不需要的调试信息，以保持最终的映像最小。

我们在第13章中介绍了readelf工具，在第14章中讲解了如何使用readelf检测ELF文件中的调试信息。代码清单15-1是用readelf读取一个ARM体系结构的网络服务应用程序的分析结果。

代码清单15-1 ELF文件调试信息

```
$ xscale_be-readelf -S websdemo
There are 39 section headers, starting at offset 0x3dfd0:

Section Headers:
[Nr] Name                Type           Addr          Off           Size      ES Flg Lk Inf Al
[ 0]                     NULL           00000000      000000      000000      00          0 0 0
[ 1] .interp                PROGBITS       00008154      000154      000013      00          A 0 0 1
[ 2] .note.ABI-tag          NOTE           00008168      000168      000020      00          A 0 0 4
[ 3] .note.numapolicy       NOTE           00008188      000188      000074      00          A 0 0 4
[ 4] .hash                  HASH           000081fc      0001fc      00022c      04          A 5 0 4
[ 5] .dynsym                DYNSYM         00008428      000428      000460      10          A 6 1 4
[ 6] .dynstr                STRTAB         00008888      000888      000211      00          A 0 0 1
[ 7] .gnu.version            VERSYM         00008a9a      000a9a      00008c      02          A 5 0 2
[ 8] .gnu.version_r          VERNEED        00008b28      000b28      000020      00          A 6 1 4
[ 9] .rel.plt               REL            00008b48      000b48      000218      08          A 5 11 4
[10] .init                  PROGBITS       00008d60      000d60      000018      00          AX 0 0 4
[11] .plt                   PROGBITS       00008d78      000d78      000338      04          AX 0 0 4
[12] .text                  PROGBITS       000090b0      0010b0      019fe4      00          AX 0 0 4
[13] .fini                  PROGBITS       00023094      01b094      000018      00          AX 0 0 4
[14] .rodata                PROGBITS       000230b0      01b0b0      0023d0      00          A 0 0 8
[15] .ARM.extab             PROGBITS       00025480      01d480      000000      00          A 0 0 1
[16] .ARM.exidx             ARM_EXIDX      00025480      01d480      000008      00          AL 12 0 4
[17] .eh_frame_hdr          PROGBITS       00025488      01d488      00002c      00          A 0 0 4
[18] .eh_frame              PROGBITS       000254b4      01d4b4      00007c      00          A 0 0 4
[19] .init_array             INIT_ARRAY     0002d530      01d530      000004      00          WA 0 0 4
[20] .fini_array             FINI_ARRAY     0002d534      01d534      000004      00          WA 0 0 4
[21] .jcr                   PROGBITS       0002d538      01d538      000004      00          WA 0 0 4
[22] .dynamic                DYNAMIC        0002d53c      01d53c      0000d0      08          WA 6 0 4
[23] .got                   PROGBITS       0002d60c      01d60c      000118      04          WA 0 0 4
[24] .data                  PROGBITS       0002d728      01d728      0003c0      00          WA 0 0 8
[25] .bss                   NOBITS         0002dae8      01dae8      0001c8      00          WA 0 0 4
[26] .comment                PROGBITS       00000000      01dae8      000940      00          0 0 1
[27] .debug_aranges          PROGBITS       00000000      01e428      0004a0      00          0 0 8
[28] .debug_pubnames         PROGBITS       00000000      01e8c8      001aae      00          0 0 1
[29] .debug_info             PROGBITS       00000000      020376      013d27      00          0 0 1
[30] .debug_abbrev           PROGBITS       00000000      03409d      002ede      00          0 0 1
[31] .debug_line             PROGBITS       00000000      036f7b      0034a2      00          0 0 1
[32] .debug_frame            PROGBITS       00000000      03a420      003380      00          0 0 4
[33] .debug_str              PROGBITS       00000000      03d7a0      000679      00          0 0 1
[34] .note.gnu.arm.idc       NOTE           00000000      03de19      00001c      00          0 0 1
```

① 记住使用交叉版本的去除无用信息的工具，例如ppc_82xx-strip。

```

[35] .debug_ranges      PROGBITS      00000000 03de35 000018 00          0 0 1
[36] .shstrtab           STRTAB          00000000 03de4d 000183 00          0 0 1
[37] .symtab              SYMTAB          00000000 03e5e8 004bd0 10        38 773 4
[38] .strtab              STRTAB          00000000 0431b8 0021bf 00          0 0 1
Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
$

```

你可以通过代码清单15-1可以看到很多段都包含着调试信息，特别是.comment段，它的大小超过2KB字节（0x940），但是对应用程序运行而言没有任何意义。这个示例文件，包括调试信息在内，大小超过275KB字节。

```

$ ls -l websdemo
-rwxrwxr-x 1 chris chris 283511 Nov 8 18:48 websdemo

```

如果用strip命令对它进行精简，我们可以减小它的体积，从而节省目标系统的存储空间。代码清单15-2显示了调用strip之后的结果：

代码清单15-2 strip后的程序

```

$ xscale_be-strip -s -R .comment -o websdemo-stripped websdemo
$ ls -l websdemo*
-rwxrwxr-x 1 chris chris 283491 Apr 9 09:19 websdemo
-rwxrwxr-x 1 chris chris 123156 Apr 9 09:21 websdemo-stripped
$

```

这里我们从可执行文件中去除了符号调试信息和.comment段。通过命令行上的-o选项指定被剔除后的文件名。用ls -l查看，可以看到，最终精简后的websdemo不到原始大小的一半。对于更大的应用程序，用strip精简后的效果更明显。一个带调试信息的Linux内核大小超过18MB，但是用strip方式精简内核之后，最后生成内核的大小不到2MB。

该方法可以用在嵌入式的调试过程中，首先用strip对文件精简，去掉调试信息，生成的文件放到目标机上运行。在主机这边，仍然保留使用strip命令以前的文件。在目标机上用gdbserver运行strip后的文件，在主机一边用gdb运行原始文件，主机-目标机通过网络进行交叉调试。

gdbserver

通过gdbserver可以使你在主机上运行GDB，而不是在嵌入式Linux目标平台。这样配置有非常明显的好处。首先，可以利用主机上比嵌入式平台更强大的处理器、更大容量的内存和硬盘存储。另外，调试程序的源代码一般也都是存放在开发主机上，而不是嵌入式平台中。

gdbserver是一个在目标板上运行的小程序，支持远程调试开发板上的进程。它在开发板上指定的被调试的程序调用，gdbserver启动时也要指定IP地址和端口号，通过指定的端口可以监听网络上发过来的gdb连接请求。代码清单15-3 是在目标板上启动gdbserver的过程。

代码清单15-3 在目标板上启动gdbserver

```
$ gdbserver localhost:2001 websdemo-stripped
Process websdemo-stripped created; pid = 197
Listening on port 2001
```

这个特别的示例启动gdbserver，配置为监听以太网的2001端口，准备调试我们已经剥除过的websdemo-stripped程序。

在开发主机上，我们通过传递要调试的二进制程序名启动GDB。这个程序需要包括完整的符号调试信息。GDB启动以后，输入一个命令连接远端的目标板，如代码清单15-4所示。

代码清单15-4 启动远端GDB会话

```
$ xscale_be-gdb -q websdemo
(gdb) target remote 192.168.1.141:2001
Remote debugging using 192.168.1.141:2001
0x40000790 in ?? ()
(gdb) p main <<<< display address of main function
$1 = {int (int, char **)} 0x12b68 <main>
(gdb) b main <<<< Place breakpoint at main()
Breakpoint 1 at 0x12b80: file main.c, line 72.
(gdb)
```

代码清单15-4中的代码片段调用开发主机中的cross-gdb。当GDB运行时，我们输入gdb target remote命令，这条命令请求GDB初始化一个开发主机和目标板的TCP/IP连接，端口是2001。在gdbserver接受连接请求以后，它将打印如下信息：

```
Remote debugging from host 192.168.0.10
```

现在GDB已经连接到了目标板上的gdbserver进程，并准备接收来自GDB的命令。剩余的内容和在本地调试应用程序完全一样。gdbserver是一个强大的工具，允许你在调试会话中充分利用开发主机的强大资源，使用gdbserver和交叉版的GDB工具可以充分利用主机上的资源来交叉调试，只需要在目标机上运行一个较小的、相对不显眼的GDB桩（stub）和被调试的程序即可。令你感到惊奇的事实是针对ARM平台的gdbserver只有54KB。

```
root@coyote:~# ls -l /usr/bin/gdbserver
-rwxr-xr-x 1 root root 54344 Jul 23 2005 /usr/bin/gdbserver
```

有一点需要注意，也是很多邮件列表中频繁提到的问题（FAQ）。你必须在开发主机上使用被配置为交叉调试器（cross-debugger）的GDB。它是一个在开发主机上运行的二进制程序，这个程序能够解析为其他体系结构编译的二进制可执行映像。这一点很重要但却常常被忽视。不能使用传统的Red Hat Linux发行版自带的GDB调试PowerPC目标程序，你必须有一个主机和目标板GDB。

调用GDB以后，将显示该GDB的一些基本版本信息、版权以及它被编译时的配置信息，代码清单15-5列出了本书使用的GDB的实例，它是由Monta Vista软件公司提供的嵌入式Linux发行版配置的PowerPC交叉环境的一部分。

代码清单15-5 交叉调试器的信息

```
$ ppc_82xx-gdb
GNU gdb 6.0 (MontaVista 6.0-8.0.4.0300532 2003-12-24)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are welcome to change it and/or distribute copies of it under
certain conditions. Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "--host=i686-pc-linux-gnu
--target=powerpc-hardhat-linux".
(gdb)
```

注意GDB启动信息的最后几行，这几行是编译这个版本GDB的配置信息。它被配置成在i686结构的处理器、GNU/Linux环境下运行，用来调试PowerPC处理器、GNU/Linux的目标系统。在编译ppc_82xx-gdb时候，可以通过./configure --host和--target来配置gdb，将gdb编译成需要的相应主机和目标机的版本。

15.3 使用共享库进行调试

既然你已经知道了如何通过主机上的GDB和目标机上的gdbserver调用一次远程调试会话，那我们就把重点放在共享库和符号调试中，它们更加复杂。除非你的应用程序采用静态链接（使用-static选项进行链接）的方式，否则程序中的很多符号将引用程序之外的代码。显而易见的例子是包括调用标准C库的函数，如fopen、printf、malloc和memcpy。另一些例子是包括对特定应用的调用，例如jack.transport-locate()（来源于JACK低延时音频服务器），它就调用了标准C库之外的库函数。

如何有效地调试带有动态库的程序，GDB必须满足下面两方面的条件：

- 必须有相应调试版本的动态库支持；
- 必须知道动态库的路径。

如果你不知道供应库的调试版本，我们仍然可以调试程序，只不过你不会得到程序调用的库的任何调试信息。通常这是完全可以接受的，当然，除非你在开发的共享库对象是你的嵌入式项目的一部分。

回想在代码清单15-4中，我们在远程目标上调用GDB。在使用target remote命令连接GDB后，GDB响应如下两行：

```
Remote debugging using 192.168.1.141:2001
0x40000790 in ?? ()
```

第一行表明GDB已经连接到指定IP地址和端口号的目标机了。第二行表明当前的程序指令寄存器在0x40000790位置处，但是in后面出现的是??，而不是具体的符号信息，这是因为在目标机上没有Linux的动态加载库（ld-x.y.z.so）的调试版本，所以0x40000790处找不到符号信息。我们是怎样得知0x40000790处引用的是Linux的动态加载库呢？

回顾在第9章中介绍的/proc文件系统。/proc文件系统中最有用的是在每一进程目录下的

maps内容（见代码清单9-16）。从代码清单15-3得知目标机上的gdbserver进程PID为197。查看进程PID197的内存映射情况，显示结果如下代码清单15-6。

代码清单15-6 gdbserver进程的内存映射情况

```
root@coyote:~# cat /proc/197/maps
00008000-00026000 r-xp 00000000 00:0e 4852444 ./websdemo-stripped
0002d000-0002e000 rw-p 0001d000 00:0e 4852444 ./websdemo-stripped
40000000-40017000 r-xp 00000000 00:0a 4982583 /lib/ld-2.3.3.so
4001e000-40020000 rw-p 00016000 00:0a 4982583 /lib/ld-2.3.3.so
bedf9000-bee0e000 rwxp bedf9000 00:00 0 [stack]
root@coyote:~#
```

从代码清单中可以看出，websdemo-stripped占用两段内存。00008000-00026000是可读可执行的代码段，0002d000-0002e000是可读写的数据段。第三个段是Linux的动态链接可执行代码段，从0x40000000开始。GDB调用其实在动态链接/加载库的代码段的第一行，因为在应用程序的代码被执行之前，GDB首先被执行。使用交叉版的readelf工具，我们看到链接器的开始地址如下：

```
# xscale_be-readelf -S ld-2.3.3.so | grep \.text
[ 9] .text PROGBITS 00000790 000790 012c6c 00 AX 0 0 16
```

从上面的数据00000790，代码段从偏移地址00000790开始，所以ld-2.3.3.so代码段被映射到内存中的起始地址为：40000000 + 00000790 = 40000790，GDB引用的地址其实就在ld-2.3.3.so（Linux的动态链接器，动态加载器）中，并且是第一条指令。

在主机上执行的是交叉版的readelf命令：xscale_be-readelf -S ld-2.3.3.so，因此要求ld-2.3.3.so必须是针对XScale体系结构的动态库。

GDB 中的共享库事件

GDB还可以显示共享库的事件信息。这对理解应用程序行为和Linux加载器行为非常有帮助。GDB还可以在共享库中设置断点，单步调试。代码清单15-7就是如何调试共享库的例子。在调试的过程中将显示共享库的完整路径。（为了便于理解代码清单中的内容作了注释。）

代码清单15-7 出现共享库事件停止GDB

```
$ xscale_be-gdb -q websdemo
(gdb) target remote 192.168.1.141:2001
Remote debugging using 192.168.1.141:2001
0x40000790 in ?? ()
(gdb) i shared <<<Display loaded shared libs
No shared libraries loaded at this time.
(gdb) b main <<<Break at main
Breakpoint 1 at 0x12b80: file main.c, line 72.
(gdb) c
Continuing.

Breakpoint 1, main (argc=0x1, argv=0xbec7fdc4) at main.c:72
```

```

72             int localvar = 9;
(gdb) i shared
From          To          Syms Read  Shared Object Library
0x40033300    0x4010260c  Yes       /opt/mvl/.../lib/tls/libc.so.6
0x40000790    0x400133fc  Yes       /opt/mvl/.../lib/ld-linux.so.3
(gdb) set stop-on-solib-events 1
(gdb) c
Continuing.
Stopped due to shared library event
(gdb) i shared
From          To          Syms Read  Shared Object Library
0x40033300    0x4010260c  Yes       /opt/mvl/.../lib/tls/libc.so.6
0x40000790    0x400133fc  Yes       /opt/mvl/.../lib/ld-linux.so.3
0x4012bad8    0x40132104  Yes       /opt/mvl/.../libnss_files.so.2
(gdb)

```

调试刚开始,当然还没有共享库被加载进来,你可以使用*i shared*命令看到这点。*i shared*命令可以列出已经加载的共享库。在应用程序的*main()*函数中设置断点,可以看到有两个共享库被加载进来,分别是Linux的动态链接/加载库和标准C库组件*libc*。

在这里,输入*stop-on-solib-event*命令,然后继续执行程序。当应用程序试图执行来自另一个共享库中的函数时,那个库将被载入。为避免迷惑,程序执行到*gethostbyname()*函数时候就会停下来,显示新加载的共享库对象信息。

这个例子说明了重要的交叉开发的概念。在目标机上运行的二进制应用程序(ELF映像),当运行到引用外部符号的代码时,需要动态加载共享库来解释该符号。通过*ldd*(参考第11章)命令,可以查看ELF文件依赖的共享库。代码清单15-8是从目标板调用*ldd*后的输出结果。

代码清单15-8 目标机上运行*ldd*的结果

```

root@coyote:/workspace# ldd websdemo
      libc.so.6 => /lib/tls/libc.so.6 (0x40020000)
      /lib/ld-linux.so.3 (0x40000000)
root@coyote:/workspace#

```

注意,目标机用*ldd websdemo*显示*websdemo*依赖的共享库时候,出现的动态库都是绝对路径,存放在根文件系统的*/lib*目录下,该路径是从目标机的根目录开始。用GDB在主机进行调试的时候,主机是无法根据这些路径找到共享库的,因为主机上相同目录*/lib*下存放的是不同体系结构的共享库,比如主机是x86体系结构的,那么*/lib*路径下存放的是针对x86体系结构的共享库,不能用于目标机ARM XScale。主机上用目标机上的共享库,所用的路径必须是相对于主机的根目录开始的绝对路径。

在主机查看*websdemo*依赖的共享库,要用交叉版本的*ldd*,通过*xscale_be-ldd*命令,可以看到共享库的路径存放在已经配置好的工具链目录中。工具链必须知道存放在主机中的共享库路径^①。代码清单15-9是在主机上运行交叉*ldd*显示的结果。

① 可以将这些共享库存放路径信息通过相应的选项或者环境变量设置传递给编译器、链接器、加载器,但是一个优秀的嵌入式开发工具发行版都应该事先就将这些路径信息传递给工具链,开发人员利用这些工具开发就方便很多。

代码清单15-9 在主机上执行ldd

```
$ xscale_be-ldd websdemo
libc.so.6 => /opt/mvl/.../xscale_be/target/lib/libc.so.6 (0xdead1000)
ld-linux.so.3 => /opt/mvl/.../xscale_be/target/lib/ld-linux.so.3 (0xdead2000)
$
```

交叉工具链应该事先就配置好库的位置信息。不仅主机的交叉gdb要知道库存放的路径，交叉编译器和交叉链接器也必须知道这些库信息^①。通过在gdb中solib-absolute-prefix命令，可以知道预先指定的库位置信息：

```
(gdb) show solib-absolute-prefix
Prefix for loading absolute shared library symbol files is
"/opt/mvl/pro/devkit/arm/xscale_be/target".
(gdb)
```

通过GDB命令set solib-absolute-prefix和set solib-search-path可以重新设置GDB查找共享库的路径信息。如果用户自己开发共享库，并且应用程序用到自开发的共享库，需要通过solib-search-path指定共享库的查找路径。详细的命令使用方法可以参考GDB命令手册。

继续对ldd的结果进行分析，从代码清单15-8和代码清单15-9可以知道共享库关联的地址。ldd命令显示共享库被Linux动态加载器加载时候的代码段的起始地址。为了更详细地看到这些地址信息，可以待目标系统的应用程序将引用到的共享库完全加载之后，用cat /proc/<pid> /maps列出该进程的内存地址映射情况。代码清单15-10显示websdemo进程完全加载共享库后的内存布局。

代码清单15-10 目标机/proc/<pid>/maps显示的内存布局

```
root@coyote:~# cat /proc/197/maps
00008000-00026000 r-xp 00000000 00:0e 4852444 /workspace/websdemo-stripped
0002d000-0002e000 rw-p 0001d000 00:0e 4852444 /workspace/websdemo-stripped
0002e000-0005e000 rwxp 0002e000 00:00 0 [heap]
40000000-40017000 r-xp 00000000 00:0a 4982583 /lib/ld-2.3.3.so
40017000-40019000 rw-p 40017000 00:00 0
4001e000-4001f000 r--p 00016000 00:0a 4982583 /lib/ld-2.3.3.so
4001f000-40020000 rw-p 00017000 00:0a 4982583 /lib/ld-2.3.3.so
40020000-4011d000 r-xp 00000000 00:0a 4982651 /lib/tls/libc-2.3.3.so
4011d000-40120000 ---p 000fd000 00:0a 4982651 /lib/tls/libc-2.3.3.so
40120000-40124000 rw-p 000f8000 00:0a 4982651 /lib/tls/libc-2.3.3.so
40124000-40126000 r--p 000fc000 00:0a 4982651 /lib/tls/libc-2.3.3.so
40126000-40128000 rw-p 000fe000 00:0a 4982651 /lib/tls/libc-2.3.3.so
40128000-4012a000 rw-p 40128000 00:00 0
4012a000-40133000 r-xp 00000000 00:0a 4982652 /lib/tls/libnss_files-2.3.3.so
40133000-4013a000 ---p 00009000 00:0a 4982652 /lib/tls/libnss_files-2.3.3.so
4013a000-4013b000 r--p 00008000 00:0a 4982652 /lib/tls/libnss_files-2.3.3.so
4013b000-4013c000 rw-p 00009000 00:0a 4982652 /lib/tls/libnss_files-2.3.3.so
becaa000-becbf000 rwxp becaa000 00:00 0 [stack]
root@coyote:~#
```

① 交叉编译器也必须知道体系结构相关的头文件位置信息。

将代码清单15-10和代码清单15-8结合起来分析, 可以看到ldd显示的共享库地址信息和/proc文件系统显示的地址信息是完全吻合的。代码清单15-8显示ldd查看共享库地址信息: Linux加载器的代码段的起始地址是0x40000000, 标准C库代码段的起始地址0x40020000。代码清单15-10中有一行为40000000-40017000 r-xp 00000000 00:0a 4982583 /lib/ld-2.3.3.so, 表明加载器的代码段加载到40000000位置, 和ldd结果分析一样。40020000-4011d000 r-xp 00000000 00:0a 4982651 /lib/tls/libc-2.3.3.so表明标准C库的代码段加载到40020000位置, 和ldd结果一致。代码清单15-9是利用交叉ldd命令显示共享库的加载地址0xdead1000和0xdead2000, 这结果表明该共享库不能在主机上加载(因为这些库是ARM体系结构的), 主机是x86体系结构的。

15.4 多任务调试

在处理多线程执行的过程中, 开发人员一般要面对两种不同的调试方案。进程可能存在于自己的地址空间, 或者与执行中的其他线程共享一个地址(以及其他系统资源)。前一个进程(不共享通用地址空间的独立进程)必须使用单独的调试会话进行调试。在目标系统上使用gdbserver调试多进程是可以的, 在开发主机上调用一个独立的GDB开启一个调试会话, 调试全部进程, 而不是调试独立的进程。

15.4.1 多进程的调试

用GDB调试多进程程序的时候, 遇到fork()系统调用^①之后, 将生成子进程, GDB对此有两种处理方式。第一种是GDB继续控制和调试父进程。第二种是GDB停止对父进程的调试, 转而调试子进程。通过set follow-fork-mode命令可以控制GDB是采用何种方式调试, follow-fork-mode有两个选择: follow child和follow parent。GDB默认的行为是follow parent, 也就是GDB默认是继续调试父进程。

代码清单15-11是一个多进程程序的部分源代码。

代码清单15-11 使用fork()产生子进程

```
...
for( i=0; i<MAX_PROCESSES; i++ ) {
    /* Creating child process */
    pid[i] = fork();                /* Parent gets non-zero PID */
    if ( pid[i] == -1 ) {
        perror("fork failed");
        exit(1);
    }
    if ( pid[i] == 0 ) {            /* Indicates child's code path */
        worker_process();          /* The forked process calls this */
    }
}
```

① 这里用到了术语系统调用, fork()是C库函数, 在C库中调用到Linux的sys_fork()系统调用。上文直接说fork()是系统调用, 应该是fork()最终调用到sys_fork()系统调用。

```

/* Parent's main control loop */
while ( 1 ) {
...
}

```

该程序在for循环里面调用fork(),产生MAX_PROCESSES个子进程,每一个新生成的子进程都要执行worker_process()函数。GDB能够知道新进程的创建,并且打印相应的信息,但是GDB继续调试父进程。代码清单15-12是GDB调试该程序的过程:

代码清单15-12 GDB在follow parent模式下的调试

```

(gdb) target remote 192.168.1.141:2001
0x40000790 in ?? ()
(gdb) b main
Breakpoint 1 at 0x8888: file forker.c, line 104.
(gdb) c
Continuing.
[New Thread 356]
[Switching to Thread 356]

Breakpoint 1, main (argc=0x1, argv=0xbe807dd4) at forker.c:104
104      time(&start_time);
(gdb) b worker_process
Breakpoint 2 at 0x8784: file forker.c, line 45.
(gdb) c
Continuing.
Detaching after fork from child process 357.
Detaching after fork from child process 358.
Detaching after fork from child process 359.
Detaching after fork from child process 360.
Detaching after fork from child process 361.
Detaching after fork from child process 362.
Detaching after fork from child process 363.
Detaching after fork from child process 364.

```

从以上代码清单看到一共生成了8个子进程, PID号从357到364。父进程的PID为356。在worker_process()处设置断点, worker_process()是每个子进程调用fork()生成之后要执行的例程。继续执行, 可以看到每一个新生成的子进程都被GDB分离了, GDB跟踪不到worker_process()断点处, 因为GDB和父进程绑定在一起, 和子进程都分离了, 所以GDB不能在worker_process()断点处停下。

如果要调试每一个子进程, 那么需要再开启新的GDB会话, 将新的GDB和生成的子进程捆绑到一起。本章最后将介绍一个非常有用的调试技术, 通过在子进程中sleep(), 留出足够的时间让新开启的GDB和子进程绑定。如何将GDB绑定到新的进程请参考15.5.2节。

如果只需要调试子进程, 在父进程调用fork()之前, 设置follow-fork-mode为follow child模式, 代码清单15-13显示了该过程。

代码清单15-13 GDB在follow child模式的调试

```

(gdb) target remote 192.168.1.141:2001

```

```

0x40000790 in ?? ()
(gdb) set follow-fork-mode child
(gdb) b worker_process
Breakpoint 1 at 0x8784: file forker.c, line 45.
(gdb) c
Continuing.
[New Thread 401]
Attaching after fork to child process 402.
[New Thread 402]
[Switching to Thread 402]

Breakpoint 1, worker_process () at forker.c:45
45      int my_pid = getpid();
(gdb) c
Continuing.

```

可以看到父进程的PID为401, 当fork()调用之后, 生成的子进程PID为402。因为设置了GDB模式为follow child模式, 所以GDB切换到PID为402的进程, 和子进程绑定在一起, 同时与父进程分离。GDB于是控制第一个子进程, 在worker_process()处的断点停下来, 等待下一步的调试命令。与此同时, 父进程继续执行, 其他的子进程被生成并且运行直到它们结束。

总结: 用上述的方式使用GDB调试程序, 每次只能调试一个进程。在遇到fork()调用的时候, 必须决定GDB是调试父进程还是调试子进程。如果要同时调试多个进程, 可以开启多个GDB, 让GDB绑定到需要调试的进程, 分别对这些进程进行调试。

15.4.2 多线程应用程序的调试

如果多线程应用程序使用的是POSIX线程库, 那么GDB是可以调试这种多线程程序的。现在NPTL库(The Native Posix Thread Library)已经成为Linux系统事实上的标准线程库。在嵌入式Linux系统中, 也使用NPTL进行多线程开发。后面的内容就用NPTL作为Linux下的标准线程库来进行讲解。

本节将采用具有如下功能的多线程演示用于调试。首先在主进程中循环调用pthread_create()产生线程, 每个产生的线程在屏幕上显示一段信息, 然后睡眠(sleep)预先定义好的时间后就退出。主进程在循环之后只是等待终端输入一个字符终止自己。代码清单15-14是目标机上运行该演示程序的结果。

代码清单15-14 多线程程序在目标机上的运行

```

root@coyote:/workspace # gdbserver localhost:2001 ./tdemo
Process ./tdemo created; pid = 671
Listening on port 2001
Remote debugging from host 192.168.1.10
^^^^ Previous three lines displayed by gdbserver

tdemo main() entered: My pid is 671
Starting worker thread 0
Starting worker thread 1
Starting worker thread 2
Starting worker thread 3

```


在前面的例子中，gdbserver准备运行，并等待来自主机的交叉GDB的连接。当GDB连接后，gdbserver通过Remote debugging...消息告知连接情况。现在我们可以执行主机上的GDB，并进行连接了。代码清单15-15显示这一会话的前半部分。

代码清单15-15 在主机上用GDB调试多线程程序

```
$ xscale_be-gdb -q tdemo
(gdb) target remote 192.168.1.141:2001
0x40000790 in ?? ()
(gdb) b tdemo.c:97
Breakpoint 1 at 0x88ec: file tdemo.c, line 97.
(gdb) c
Continuing.
[New Thread 1059]
[New Thread 1060]
[New Thread 1061]
[New Thread 1062]
[New Thread 1063]
[Switching to Thread 1059]

Breakpoint 1, main (argc=0x1, argv=0xbffffdd4) at tdemo.c:98
98          int c = getchar();
(gdb)
```

现在连接到了目标（代码清单15-14中Remote debugging...消息），在传进新线程循环之后设置一个断点，并继续调试。当新线程产生以后，GDB显示一条创建新线程的ID信息。线程1059是tdemo主线程，主线程执行main()函数体中的代码。线程1060~1063是调用pthread_create()而产生的新线程。

当GDB遇到断点后，它会显示一条消息：[Switching to Thread 1059]，以此说明这是执行后遇到断点的那个线程。这是调试会话中处于激活状态的线程，具体参考GDB文档中“current thread”部分。

GDB可以在几个线程中切换来进行调试。可以设置附加的断点、检测数据、显示调用信息等。还可以在当前线程的私有单个栈帧中工作。代码清单15-16是GDB调试多线程的操作序列，接着代码清单15-15继续调试。

代码清单15-16 GDB调试多线程

```
...
(gdb) c
Continuing.

<<< Ctl-C to interrupt program execution
Program received signal SIGINT, Interrupt.
0x400db9c0 in read () from /opt/mvl/.../lib/tls/libc.so.6
(gdb) i threads
5 Thread 1063 0x400bc714 in nanosleep ()
  from /opt/mvl/.../lib/tls/libc.so.6
4 Thread 1062 0x400bc714 in nanosleep ()
  from /opt/mvl/.../lib/tls/libc.so.6
```

```

3 Thread 1061 0x400bc714 in nanosleep ()
  from /opt/mv1/.../lib/tls/libc.so.6
2 Thread 1060 0x400bc714 in nanosleep ()
  from /opt/mv1/.../lib/tls/libc.so.6
* 1 Thread 1059 0x400db9c0 in read ()
  from /opt/mv1/.../lib/tls/libc.so.6
(gdb) thread 4          <<< Make Thread 4 the current thread
[Switching to thread 4 (Thread 1062)]
#0 0x400bc714 in nanosleep ()
  from /opt/mv1/.../lib/tls/libc.so.6
(gdb) bt
#0 0x400bc714 in nanosleep ()
  from /opt/mv1/.../lib/tls/libc.so.6
#1 0x400bc4a4 in __sleep (seconds=0x0) at sleep.c:137
#2 0x00008678 in go_to_sleep (duration=0x5) at tdemo.c:18
#3 0x00008710 in worker_2_job (random=0x5) at tdemo.c:36
#4 0x00008814 in worker_thread (threadargs=0x2) at tdemo.c:67
#5 0x40025244 in start_thread (arg=0xfffffdcf) at pthread_create.c:261
#6 0x400e8fa0 in clone () at ../sysdeps/unix/sysv/linux/arm/clone.S:82
#7 0x400e8fa0 in clone () at ../sysdeps/unix/sysv/linux/arm/clone.S:82
(gdb) frame 3
#3 0x00008710 in worker_2_job (random=0x5) at tdemo.c:36
36      go_to_sleep(random);

(gdb) 1                  <<< Generate listing of where we are
31      }
32
33      static void worker_2_job(int random)
34      {
35          printf("t2 sleeping for %d\n", random);
36          go_to_sleep(random);
37      }
38
39      static void worker_3_job(int random)
40      {
(gdb)

```

调试过程中有几个细节必须要注意：GDB本身会为每个线程分配一个id号（从1开始，1、2、3...），GDB引用这些id来操作相应的线程。当在一个线程中运行时碰到断点，所有的其他线程都被暂停。GDB用（*）标识当前运行的线程。在设置断点的时候，可以在每个线程的私有代码段设置唯一的断点。如果在所有线程都要运行的公共代码部分处设置断点，那么调试时无法判断哪个线程先遇到断点。

本章最后的参考文档“GDB用户手册”包括更详细的多线程调试方法。

15.4.3 引导装入程序/闪存代码的调试

调试闪存上的代码是一种特有的挑战。最明显的限制是GDB和gdbserver在设置目标断点上合作的方式。在第14章讨论GDB的远程串行协议时，你学习了如何在应用程序中插入断点^①。GDB

① 参考代码清单14-5。

将断点处的指令用带有控制信息的新指令覆盖。但是在ROM或者闪存上，GDB不能覆盖指令，所以这种设置断点的方式行不通。

为了解决这个问题，现在大部分处理器都包括相当数量的调试寄存器。这些调试功能需要特定平台硬件支持和处理器支持。调试闪存和ROM代码最常用的技术是使用JTAG接口。JTAG接口支持遵循处理器硬件断点设置。详细信息请参考14.4.2节。

15.5 远程调试的附加选项

有时需要利用串行端口来进行远程调试，有时要将GDB绑定到一个正在运行的进程上。下面来介绍这些操作技巧^①。

15.5.1 串行端口调试

通过串行端口调试非常简单。当然，目标机上必须有一个串行端口，并且该串行端口没有被其他进程占用。主机也是同样的要求。当主机-目标机都有可用的串行端口，并且通过串行端口线连接两者之后，在目标机启动gdbserver，只需要用对应的串行端口设备取代前面章节介绍的IP:Port选项。

在目标机上的命令选项如下：

```
root@coyote:/workspace # gdbserver /dev/ttyS0 ./tdemo
Process ./tdemo created; pid = 698
Remote debugging using /dev/ttyS0
```

在主机上用target remote命令后面跟上相应串行端口设备，操作如下：

```
$ xscale_be-gdb -q tdemo
(gdb) target remote /dev/ttyS1
Remote debugging using /dev/ttyS1
0x40000790 in ?? ()
```

15.5.2 绑定到正在运行的进程

当一个进程正在运行的时候，可以通过将gdbserver绑定到该进程，然后用GDB检查它的状态，不用杀死该进程再重启来调试进程。下面是对应的操作步骤：

```
root@coyote:/workspace # ps ax | grep tdemo
1030 pts/0 S1+ 0:00 ./tdemo
root@coyote:/workspace # gdbserver localhost:2001 --attach 1030
Attached; pid = 1030
Listening on port 2001
```

如果检查完进程的状态，在主机上可以用detach命令将绑定的进程去掉，使用该命令会将gdbserver和调试进程分离开来，并且结束当前调试会话。该进程继续运行一直到结束。注意：当用attach将gdbserver绑定到进程的时候，该进程会暂停，等待主机发送过来的调试消息。只

^① 参考第13章。

有gdbserver收到继续运行的调试信息或者detach命令后，调试进程才会继续运行。detach命令可以在任意时候使用，用来结束调试会话，将调试进程和gdbserver之间的绑定去掉。detach之后该进程将继续在目标系统上运行。

15.6 小结

- 交叉（远程）调试时，主机运行的调试文件必须带有符号调试信息，目标机上运行的调试文件可以用strip精简去掉调试符号信息的程序。
- 目标机上运行的gdbserver将主机上运行的交叉GDB和目标机上待调试的程序连接起来，在主机上交叉调试应用程序就像在本地调试一样。
- 通常主机的GDB和目标机的gdbserver通过网络套接字连接进行通信。也可以使用串口让GDB和gdbserver通信。
- GDB在遇到共享库事件的时候可以暂停进程并且自动加载共享库。GDB等工具链应该事先就配置好相应的交叉开发所需要的共享库路径。也可以使用GDB命令手工配置查找共享库路径。
- 调试多进程程序，可以同时开启多个GDB会话，每个GDB绑定到相应的进程来进行调试。
- GDB默认的follow-fork-mode是父进程，遇到fork()，GDB继续调试父进程。也可以设置follow-fork-mode为子进程，fork()调用之后，GDB调试子进程。
- GDB可以很方便地调试遵循POSIX线程库的多线程程序。当前Linux的默认线程库是NPTL。
- GDB可以绑定到正在运行的进程，也可以分离已经绑定的进程。

参考资料

GDB: GNU工程调试器在线文档

<http://sourceware.org/gdb/onlinedocs/>

GDB Pocket Reference

Arnold Robbins

O'Reilly Media, 2005



本章内容

- Linux源代码的组织
- 为开发板定制Linux
- 平台初始化
- 汇总
- 小结

把Linux移植到一个新的硬件平台上并不困难。Linux源码树中包含了许多移植好的代码，涵盖范围超过20种体系结构以及更多的处理器。知道从哪里入手往往是最困难的地方。

本章介绍了将Linux移植到自己开发板上的基础知识，该Linux提供了基本的网络和串口控制台操作。我们从一个体系结构和平台的角度分析Linux源代码的组织，然后深入研究早期的内核初始化代码，以便理解平台初始化提供的机制。最后，我们介绍如何将Linux移植到定制的PowerPC硬件平台。

16.1 Linux 源代码的组织

不久以前，有非常多的地方^①提供了各种各样版本的Linux，其中有专注PowerPC版本的，也有专注ARM版本的，等等。虽然没必要特意这么做，但这却是必要的。将不同体系结构和功能合并到传统的内核上是要花一定时间的。另外，提供单独一套源码树意味着可以更快速地获得给定的体系结构的最新特性。

内核开发人员为了把各种体系结构都统一到一个通用的源码树中，已经做了大量的工作。除了极少数例外，现在使用的案例基于Linux 2.6源代码。你可以从www.kernel.org下载并为各种处理器和工业标准参考版编译可以工作的内核。

体系结构分支

在第4章中，我们介绍了Linux内核源码树的全面的结构，本章将主要说明Linux内核源代码

^① 地方（homes）是指公共的源代码仓库，例如因特网上的一台服务器。

中和体系结构相关的分支。代码清单16-1列出了最近内核快照的`../arch`目录下的内容。正如第4章指出的，`../arch`子目录在容量上是第二大的。在一个最近的Linux发行版中，在文件数方面，该目录是最多的（除了`../include`目录以外），只有`../driver`子目录中的文件比它多。

代码清单16-1 Linux内核`../arch`目录

```
[chris@pluto linux]$ ls ../arch
alpha  cris  i386  m68k  parisc  s390  sparc  v850
arm     frv   ia64  m68knommu  ppc     sh     sparc64  x86_64
arm26   h8300  m32r  mips  ppc64   sh64   um      xtensa
```

通过这个清单可以看到，Linux内核支持了24种不同的体系结构，我们把每一个都当作一种体系结构分支（architecture branch），以方便讨论。

每种体系结构分支都有一些共同的组成部分，例如每种体系结构的顶层目录都包含`Kconfig`文件，如第4章中`Kconfig`驱动程序内核配置工具。当然，每种体系结构分支的顶层目录还有相应的`makefile`文件。所有体系结构分支的顶层目录都含有`kernel`子目录，因为有些内核特性是和体系结构相关的。除了两种平台以外，其他平台都含有`mm`子目录，这里存放了和体系结构相关的内存管理代码。

多数体系结构分支中包含了一个`boot`子目录，该目录常用于创建（通过自己的`makefile`）一个特定的可启动的目标。有些分支还包含了`mach-*`这样的子目录，这个目录下用来存放和特定机器或硬件平台相关的代码。另一个在体系结构分支下频繁出现的子目录是`configs`，这个目录下存放了许多流行的体系结构和每个所支持的硬件平台的默认配置。

在本章后面的内容中，我们在PowerPC体系结构上展开讨论和举例。PowerPC是最流行的体系结构之一，支持多种处理器和开发板。代码清单16-2列出了最近的Linux内核发行版PowerPC分支中`../arch/ppc`下`configs`目录的内容。

代码清单16-2 PowerPC `configs`目录内容

```
[chris@pluto linux]$ ls ../arch/ppc/configs/
ads8272_defconfig  IVMS8_defconfig  prpmc750_defconfig
apus_defconfig     katana_defconfig  prpmc800_defconfig
bamboo_defconfig   lite5200_defconfig  radstone_defconfig
bseip_defconfig    lopec_defconfig    redwood5_defconfig
bubinga_defconfig  luan_defconfig     redwood6_defconfig
chestnut_defconfig mbx_defconfig       rpx8260_defconfig
common_defconfig   mpc834x_sys_defconfig  rpxcllf_defconfig
cpci405_defconfig  mpc8540_ads_defconfig  rpxlite_defconfig
cpci690_defconfig  mpc8548_cds_defconfig  sandpoint_defconfig
ebony_defconfig    mpc8555_cds_defconfig  spruce_defconfig
ep405_defconfig    mpc8560_ads_defconfig  stx_gp3_defconfig
est8260_defconfig  mpc86x_ads_defconfig  sycamore_defconfig
ev64260_defconfig  mpc885ads_defconfig   TQM823L_defconfig
ev64360_defconfig  mvme5100_defconfig    TQM8260_defconfig
FADS_defconfig     ocotea_defconfig      TQM850L_defconfig
gemini_defconfig   pmac_defconfig        TQM860L_defconfig
```

hdpu_defconfig	power3_defconfig	walnut_defconfig
ibmchrp_defconfig	pplus_defconfig	

在PowerPC体系结构分支中，configs目录下每一个这样的入口点都表示针对该硬件平台的特定移植。例如，walnut_defconfig文件定义了针对AMCC Walnut PPC405评估板的默认配置选项。mpc8540_ads_defconfig文件描述了Freescale MPC8540 ADS评估板的默认配置选项。如第4章所述，为这些参考平台创建内核，你首先要使用这些默认的配置文件来配置你的内核源码树，如下所示：

```
$ make ARCH=ppc CROSS_COMPILE=ppc_85xx- mpc8540_ads_defconfig
```

这个make命令（来自Linux目录顶层的）使用Freescale MPC8540 ADS评估板的默认配置文件配置内核源码树。

Linux内核源码树并没有完成统一，其中的一个原因是每种体系结构处理平台相关文件的方式不同。在PowerPC分支中，你可以找到一个包含平台相关的代码的目录platforms。浏览这个目录，你可以看到许多源文件是以各自硬件平台命名的。在.../arch/ppc/platforms目录下也有几个子目录用于特定的PowerPC变量。

与之类似，ARM分支包含一系列mach-*目录，每一个目录代表一种特定的硬件平台，而MIPS分支也有一套以指定平台命名的子目录。

16.2 为开发板定制 Linux

在第7章中将U-Boot移植到一个新硬件平台时，我们找到了和我们的新开发板最为贴近的配置，并基于该配置进行移植。我们使用类似的技术将Linux移植到新平台，假定所选定的处理器已经在内核中得到了支持。将新处理器移植到Linux具有极大的挑战性，而且超出本书的范围。

我们的移植Linux工作选择了基于Freescale MPC5200 32位嵌入式处理器的PowerPC开发板。通过浏览最近的Linux发行版的默认配置目录（如代码清单16-2所示），可以找到包含MPC5200处理器的配置文件。因为它是唯一支持该处理器的配置，所以我们就以它为工作的起点。

我们做的这个练习所使用的硬件平台由联合电气协会（United Electronic Industries）友情提供。这块板子叫作PowerDNA Controller，它有一个简单的结构图，包括板上闪存、动态RAM、一个串行接口和几个I/O设备，其中大部分都已经集成到MPC5200处理器内部。图16-1是PowerDNA Controller的功能框图。

16.2.1 前提和假设

当Linux内核得到来自引导装入程序传递过来的控制权时，内核将做一些基本的假设。其中最重要的一个是假设引导装入程序必须初始化DRAM控制器。Linux不会参与芯片级SDRAM控制器初始化，因此Linux假定系统RAM已经存在并完全可用。我们的PowerDNA Controller使用的U-Boot将初始化CPU、DRAM和其他相关的硬件，以满足最小系统操作的需求。

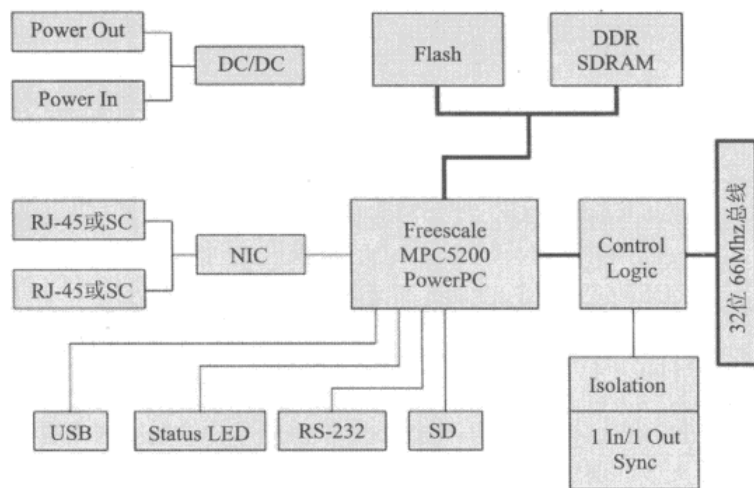


图16-1 UEI PowerDNA Controller开发板

引导装入程序还应该初始化系统内存映射。通常这是通过一系列处理器寄存器完成的，这些寄存器定义了片选信号在哪些给定的内存地址范围内处于激活状态。在Freescale MPC5200用户指南的第3章中描述了完成这个任务的寄存器。

引导装入程序可能还要完成附加的和硬件相关的初始化任务。在一些开发板中，内核假定串口已经配置好，即可以通过串口显示早期的内核启动信息，这些信息远早在内核自身的串口驱动程序安装以前执行。一些体系结构和硬件平台也都提供了类似*_serial_putc()的函数，该函数可以向被引导装入程序或一些早期内核安装代码配置过的串口发送字符串。可以使用grep命令在PowerPC体系结构中搜索CONFIG_SERIAL_TEXT_DEBUG找到这个例子。

总之，将Linux移植到新平台的基本前提是，引导装入程序已经被移植并安装在我们的开发板上，并且任何与板极相关的底层硬件初始化都已经完成。但是这里没必要初始化Linux直接支持的设备，例如以太网控制器或I2C控制器，内核自然会处理它们的。

为最贴近你自己的开发板配置和构建Linux内核是一个好主意。它会帮助你配置Linux内核源码树，而且编译过程中不会出现错误，这是我们移植工作很好的开端。回忆第5章中编译Linux 2.6内核使用的命令：

```
$ make ARCH=ppc CROSS_COMPILE=ppc_82xx- uImage
```

这个命令将生成一个和U-Boot兼容的Linux可启动映像，这由uImage选项指定。

16.2.2 定制内核初始化

既然已经知道了基本的内核源码树的起始点，那就来确定从哪里开始定制我们特殊的开发板。前面提到，对PowerPC体系结构来说，其板极相关的文件存在于../arch/ppc/platforms目录中。当然这并不是严格要求的，但是如果你想要把你的补丁提交给Linux内核开发社区并得

到认可，就必须要考虑正确的格式和一致性的问题。

我们在platforms目录中看到一个名为lite5200.c的文件。该文件非常简单，包括两个数据结构和五个函数，代码清单16-3给出该文件的函数。

代码清单16-3 5200平台文件的函数

```
lite5200_show_cpuinfo() /* Prints user specified text string */
lite5200_map_irq()      /* Sets h/w specific INT logic routing */
lite5200_setup_cpu()    /* CPU specific initialization */
lite5200_setup_arch()   /* Arch. specific initialization */
platform_init()         /* Machine or board specific init */
```

下面看看这些函数是怎么使用的。第5章曾简要地解释了底层内核初始化，这里我们针对特殊的体系结构深入看一下。虽然各体系结构之间的细节不尽相同，但是当你掌握其中一个以后，其他的就很容易学习了。

我们在第5章看到了系统加电后的早期控制流，引导装入程序把控制权传递给内核第二阶段引导装入程序（bootstrap loader），这是通过内核中head.o模块传递给Linux内核的，这里是平台相关的初始化开始的地方。代码清单16-4给出了.../arch/ppc/kernel/head.S文件中部分代码。

代码清单16-4 调用早期机器初始化

```
...
/*
 * Do early bootinfo parsing, platform-specific initialization,
 * and set up the MMU.
 */
    mr    r3,r31
    mr    r4,r30
    mr    r5,r29
    mr    r6,r28
    mr    r7,r27
    bl    machine_init
    bl    MMU_init
...
```

这里可以看到用汇编语言调用的machine_init，具有特殊含义的是寄存器r3~r7的操作。这些寄存器被预期用来存放熟悉的值，很快你就会看到。在启动最初期，它们被存储在PowerPC的通用寄存器r27~r31中。这里寄存器r3~r7会从先前保存的值中恢复回来。

machine_init()函数在setup.c文件中定义，该文件也在体系结构相关的kernel目录中：.../arch/ppc/kernel/setup.c。代码清单16-5列出该文件的入口函数的起始点。

代码清单16-5 setup.c中的machine_init()函数

```
void __init
machine_init(unsigned long r3, unsigned long r4, unsigned long r5,
             unsigned long r6, unsigned long r7)
{
#ifdef CONFIG_CMDLINE
    strcpy(cmd_line, CONFIG_CMDLINE, sizeof(cmd_line));
#endif /* CONFIG_CMDLINE */
```

```

#ifdef CONFIG_6xx
    ppc_md.power_save = ppc6xx_idle;
#endif
#ifdef CONFIG_POWER4
    ppc_md.power_save = power4_idle;
#endif

platform_init(r3, r4, r5, r6, r7);

if (ppc_md.progress)
    ppc_md.progress("id mach(): done", 0x200);
}

```

在这个简单的函数中有一些非常有用的知识。首先，注意`machine_init()`的参数代表PowerPC通用寄存器`r3~r7`^①。这些寄存器是在汇编语言调用`machine_init`之前被初始化的。正如在代码清单16-5中看到的，这些寄存器的值未经修改就传给了`platform_init()`。我们需要根据我们的平台修改这个函数。（马上就对其展开讨论。）

代码清单16-5还包含了一些和体系相关的电源管理函数的调用。如果你的内核配置为支持PowerPC 6xx（.config文件中定义了`CONFIG_6xx`），那么结构中 will 填充一个指向体系相关的电源管理函数（`ppc6xx_idle`）。与之类似，如果你的内核配置为PowerPC G5核（`CONFIG_POWER4`），那么在同样的结构成员中将填充一个指向体系相关的电源管理程序。16.3.3节描述了该结构。

16.2.3 静态内核命令行

在代码清单16-5中，`machine_init()`函数的一个最令人感兴趣的操作，是保存默认内核命令行。在内核配置中，如果配置了`CONFIG_CMDLINE`选项，那么这个操作即被开启。在一些平台中，引导装入程序不支持内核命令行。在这些情况下，内核命令行可以静态编译到内核中。图16-2说明了该操作的配置选项。

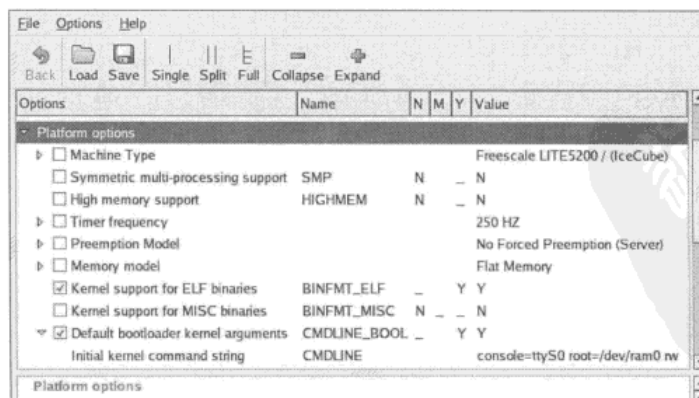


图16-2 默认内核命令行

① 按照惯例，这些PowerPC寄存器参数按照C语言的方式进行传递。

选中图16-2中的Default bootloader kernel arguments，并根据图所示编辑Initial kernel command string选项。这个操作会在.config文件中生成一些记录，如代码清单16-6所示。

代码清单16-6 默认内核命令行的配置

```
...
CONFIG_CMDLINE_BOOL=y
CONFIG_CMDLINE="console=ttyS0 root=/dev/ram0 rw"
...
```

代码清单16-6中的省略号表示我们只是摘取了.config文件的一个小片段。当内核编译系统处理这些配置符号时，它们会成为.../include/linux/autoconf.h文件中的记录项，如代码清单16-7所示。

代码清单16-7 默认内核命令行在autoconf.h文件中的选项

```
...
#define CONFIG_CMDLINE_BOOL 1
#define CONFIG_CMDLINE "console=ttyS0 root=/dev/ram0 rw"
...
```

现在回到代码清单16-5，来看看下面的命令行：

```
strcpy(cmd_line, CONFIG_CMDLINE, sizeof(cmd_line));
```

可以看到，这个内核级的字符串复制函数把CONFIG_CMDLINE定义的字符串复制到一个全局内核变量中，该变量叫作cmd_line。这一点很重要，因为许多函数和设备驱动程序在启动初期可能需要检查内核命令行。这个全局变量cmd_line隐藏在.data段的起始部分，在.../arch/ppc/kernel/head.S汇编文件中定义。

这里有一个微妙的细节值得注意，回过头看看代码清单16-4，可以看到汇编语言调用的machine_init函数发生在MMU_init之前，这意味着任何能够在machine_init中运行的代码是在内存访问受限的上下文中执行的。现在有很多包含MMU的处理器都不能访问没有经过处理器硬件寄存器初始映射的内存^①。典型的做法是，启动时让一小部分内存可用，为载入并解压缩内核以及ramdisk映像提供空间，试图访问这些早期限制之外的代码或数据将失败。每种体系结构和平台可能有不同的早期内存访问限制。8MB~16MB的值并不是典型的。我们必须记得任何在machine_init中执行的代码，包括我们的平台初始化在内，都发生在这个上下文中。如果在调试你的新内核移植时遇到数据访问错误（PowerPC DSI例外^②），应该马上猜想到你没有正确地映射你的代码试图访问的内存区域。

16.3 平台初始化

下面快速回顾一下早期初始化时的代码流。图16-3说明了从引导装入程序或第二阶段引导装

① AMCC PPC405是其中的佼佼者。对它感兴趣的读者不妨看看这块处理器的BAT寄存器。

② 参考本章结尾的编程环境手册参考以了解PowerPC DSI例外的细节内容。

入程序与平台相关的初始化代码执行过程。

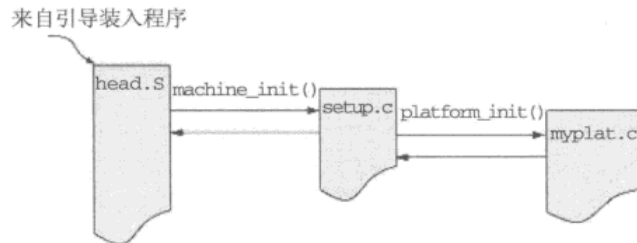


图16-3 平台初始化的控制流

head.S和setup.c文件都在PowerPC体系结构的.../arch/ppc/kernel目录下。而我们自己平台相关的文件会放在.../arch/ppc/platforms目录中。在图16-3中用myplat.c文件代表我们自己的平台文件。我们现在详细地研究这个和平台相关的初始化文件。

在代码清单16-3中，我们列出了平台相关文件lite5200.c中的函数。其中除platform_init()以外，每一个函数都被声明为静态的。所以这就是平台相关文件的入口点，如图16-3所示。文件中其余的函数只会被该文件自己引用。

我们看看入口函数platform_init()。代码清单16-8复制了来自lite5200.c文件中的platform_init()函数。

代码清单16-8 Lite5200中的platform_init()函数

```

void __init
platform_init(unsigned long r3, unsigned long r4,
              unsigned long r5, unsigned long r6,
              unsigned long r7)
{
    /* Generic MPC52xx platform initialization */
    /* TODO Create one and move a max of stuff in it.
       Put this init in the syslib */

    struct bi_record *bootinfo = find_bootinfo();

    if (bootinfo)
        parse_bootinfo(bootinfo);
    else {
        /* Load the bd_t board info structure */
        if (r3)
            memcpy((void*)&__res, (void*)(r3+KERNELBASE),
                  sizeof(bd_t));

#ifdef CONFIG_BLK_DEV_INITRD
        /* Load the initrd */
        if (r4) {
            initrd_start = r4 + KERNELBASE;

```



```

        initrd_end = r5 + KERNELBASE;
    }
#endif

/* Load the command line */
if (r6) {
    *(char *) (r7+KERNELBASE) = 0;
    strcpy(cmd_line, (char *) (r6+KERNELBASE));
}

/* PPC Sys identification */
identify_ppc_sys_by_id(mfspr(SPRN_SVR));

/* BAT setup */
mpc52xx_set_bat();

/* No ISA bus by default */
isa_io_base      = 0;
isa_mem_base     = 0;

/* Powersave */
/* This is provided as an example on how to do it. But you
   need to be aware that NAP disable bus snoop and that may
   be required for some devices to work properly, like USB
   ... */
/* powersave_nap = 1; */

/* Setup the ppc_md struct */
ppc_md.setup_arch    = lite5200_setup_arch;
ppc_md.show_cpuinfo  = lite5200_show_cpuinfo;
ppc_md.show_percpuinfo = NULL;
ppc_md.init_IRQ      = mpc52xx_init_irq;
ppc_md.get_irq       = mpc52xx_get_irq;

#ifdef CONFIG_PCI
    ppc_md.pci_map_irq = lite5200_map_irq;
#endif

ppc_md.find_end_of_memory = mpc52xx_find_end_of_memory;
ppc_md.setup_io_mappings  = mpc52xx_map_io;

ppc_md.restart      = mpc52xx_restart;
ppc_md.power_off    = mpc52xx_power_off;
ppc_md.halt         = mpc52xx_halt;

```



```

    /* No time keeper on the LITE5200 */
    ppc_md.time_init      = NULL;
    ppc_md.get_rtc_time   = NULL;
    ppc_md.set_rtc_time   = NULL;

    ppc_md.calibrate_decr = mpc52xx_calibrate_decr;
#ifdef CONFIG_SERIAL_TEXT_DEBUG
    ppc_md.progress       = mpc52xx_progress;
#endif
}

```

这个函数包括了特定开发板需要的大量的定制内容，自搜索引导装入程序提供的板极相关的数据开始。我们将在16.3.2节中讨论细节。

如果你的内核配置为初始化ramdisk (initrd)^①，那么该函数接下来将保存ramdisk映像的起始和结束地址。注意，通常情况下这两个地址通过PowerPC通用寄存器r4和r5传递。利用这些寄存器传递initrd的地址是引导装入程序的职责。接着，内核会把initrd映像从原始内存（存放引导装入程序的地址，或非易失性的Flash映像）载入到一个内部的内核ramdisk结构中。

接下来看看存储内核命令行的代码，其地址在platform_init()函数中通过寄存器r6和r7进行传递，r6和r7分别记录了起始地址和结束地址。与早期描述的静态内核命令行参数方法有所不同，这个命令行参数是通过引导装入程序使用platform_init()函数传递的，而另一种是直接编译进内核的。

复制initrd和内核命令行的代码简单易懂。基本上，引导装入程序传递的寄存器中包含了这些数据结构所在的内存地址。不过这有里面一点微妙，也许你已经想知道KERNELBASE常量的作用了吧。理解它是掌握内核启动流程的复杂部分的关键点之一。

引导装入程序所提供的地址都是物理地址，这意味着它们是数据存放在内存芯片中的真实的硬件地址。引导装入程序一般在不支持虚拟内存的情况下进行工作。但是，内核自身是静态链接到一个熟悉的、用户配置的基本地址，这个地址就是KERNELBASE。（这个值本身和讨论的内容无关，它是用户配置的，但是实际上都不会进行修改，只使用默认值0xC0000000。）

这一步是在head.s文件中比较有趣的一个地方完成的。当内核解压，重定向到RAM中时（通常定位在0地址），全部代码段和数据符号都链接到内核的虚拟地址KERNELBASE。这一点可以通过查看内核的符号映射文件得知。内核的符号映射文件在内核创建过程中产生，名为System.map^②。不过，在开启MMU之前的执行期间，物理地址就是实际的硬件地址。这意味着开启MMU之前的所有代码和虚拟内存映射都必须被重定向，并且必须修改对符号的访问。这要给符号地址增加偏移以便访问。后面用一个例子你就会明白。

16.3.1 早期变量访问

我们假设代码段在启动过程的早期需要很早就访问cmd_line变量，此时代码的运行环境是

① 第6章介绍了初始化ramdisk或initrd的内容。

② 我们在第4章介绍了System.map文件。

实地址和物理地址以1:1进行映射。正如前面指出的，这个变量在head.S文件中定义，当内核被链接以后，在.data段终止。在Linux内核的System.map文件中可以找到cmd_line链接后的地址：

```
$ cat System.map | grep cmd_line
c0115000 D cmd_line
```

如果运行在真实=物理模式（禁用MMU）下并且使用它的符号访问这个变量，那么我们将试着在3GB以上的地址空间进行读写操作。大多数较小的嵌入式系统无法映射到这个区域，因此将导致未定义错误或者崩溃。即使我们在这个地址上有物理内存，也不可能在启动过程的早期被映射并访问，所以我们不得不调整对这个变量的引用以便访问。

代码清单16-9复制了head.S文件中实现上述过程的代码片段。

代码清单16-9 变量引用修正

```
relocate_kernel:
    addis r9,r26,klimit@ha /* fetch klimit */
    lwz   r25,klimit@l(r9)
    addis r25,r25,-KERNELBASE@h
```

这个摘自PowerPC中head.S文件的代码片段很好地解释了我们描述的论点。klimit变量代表了内核映像的结束，它在其他地方定义为char *klimit，所以它是一个指向所包含的地址的指针。在代码清单16-9中，我们取到了klimit的地址，求出它和前面计算出来的偏移量的和，这个偏移量通过寄存器r26传递，并把结果保存在寄存器r9中。现在寄存器r9的高16位记录了klimit调整后的值，低16位填充0^①。klimit通过前面计算的偏移量进行的调整，通过寄存器r26传递。

在下一行中，lwz指令将寄存器r9和klimit（klimit地址的低16位）偏移加到一起，作为一个有效的地址存在寄存器r25中（记住，klimit是一个指针，我们应该对klimit指向的值感兴趣），寄存器r25此时持有存储在klimit变量中的指针。在代码清单16-9的最后一行，我们从r25中减去内核链接后的基地址（KERNELBASE），调整指向实际的物理地址。在C语言中是这样做的：

```
unsigned int *tmp;          /* represents r25 */
tmp = *klimit;
tmp -= KERNELBASE;
```

总之，我们引用了存储在klimit中的指针，调整它的值为真实的（物理）地址以便使用其中的内容。当内核打开MMU和虚地址后，我们就不用考虑内核在所链接的地址上运行的问题了，也不用管内核实际定位在哪块物理内存中。

16.3.2 开发板信息结构

许多引导装入程序用于PowerPC平台，但是目前仍然没有统一的方法传递板极相关的数据，例如串口波特率、内存大小和引导装入程序配置的其他底层硬件参数。代码清单16-8中平台相关

① 关于PPC汇编语言语法的详细内容，请参考本章末的“参考资源”。

的文件支持两种不同的方法，数据存储为bi_record结构和存储为bd_info结构^①。这两种方法产生相似的结果：硬件相关的数据从引导装入程序中传递到内核的这些结构中。

下面的代码片段取自代码清单16-8，它保存了引导装入程序支持的硬件配置。

```
struct bi_record *bootinfo = find_bootinfo();

if (bootinfo)
    parse_bootinfo(bootinfo);
else {
    /* Load the bd_t board info structure */
    if (r3)
        memcpy((void*)&__res, (void*)(r3+KERNELBASE),
               sizeof(bd_t));
}
```

首先，我们查找一个把数据结构看作bi_recode的特殊标记符。如果找到这个标记符，bootinfo指针设置为bootinfo记录的起始地址。记录在这里被解析、收集和硬件相关的数据，这一步可以在.../arch/ppc/kernel/setup.c文件中看到。通常，bi_records包含了内核命令行参数、initrd映像的起始和结束地址、机器类型和内存大小。当然，你可以根据自身需要加以扩展。

如果没有找到bi_record数据，PowerPC体系结构会认为这个数据以U-Boot板极信息结构或bd_info结构存在。创建这个数据结构并用r3寄存器传递地址是引导装入程序的职责。通常，bd_info结构可以获得很多硬件信息，包括DRAM、Flash、SRAM、处理器时钟比率、总线频率、串口波特率设置，以及更多。

bi_record结构在.../include/asm-ppc/bootinfo.h文件中，bd_info结构可以在.../include/asm-ppc/ppcboot.h文件中找到。

利用完成硬件设置可能需要的数据或与内核通信是平台相关的程序的职责。例如，platform_init()函数设置一个函数指针，该函数的名字就揭示了它的功能。这里再给出代码清单16-8中的代码：

```
ppc_md.find_end_of_memory = mpc52xx_find_end_of_memory;
```

查看.../arch/ppc/syslib/mpc52xx_setup.c中的mpc52xx_find_end_of_memory()函数，我们发现下面的代码：

```
u32 ramsize = __res.bi_memsize;

if (ramsize == 0) {
    ... /* Find it another way */
}

return ramsize;
```

① 每一种方法都有其自己的出处。bi_record结构最初在U-Boot中创建，bd_info结构试图在所有的平台中统一。这两种方法都支持多种平台。

上面的__res数据结构是板极信息结构，它的地址通过寄存器r3从引导装入程序传递给我们。可以看到，普通的安装代码存储了通过引导装入程序传递的剩余的数据（通常都这么叫），但是它们将一直等待机器或平台相关的代码使用。

16.3.3 机器相关的调用

许多内核需要的公共程序（或用于初始化，或用于操作）都是体系结构和机器（CPU）相关的。在代码清单16-8给出的platform_init()函数中，可以看到下面的代码：

```
...
/* Setup the ppc_md struct */
ppc_md.setup_arch = lite5200_setup_arch;
ppc_md.show_cpuinfo = lite5200_show_cpuinfo;
ppc_md.show_percpuinfo = NULL;
ppc_md.init_IRQ = mpc52xx_init_irq;
ppc_md.get_irq = mpc52xx_get_irq;

#ifdef CONFIG_PCI
    ppc_md.pci_map_irq = lite5200_map_irq;
#endif

ppc_md.find_end_of_memory = mpc52xx_find_end_of_memory;
ppc_md.setup_io_mappings = mpc52xx_map_io;

ppc_md.restart = mpc52xx_restart;
ppc_md.power_off = mpc52xx_power_off;
ppc_md.halt = mpc52xx_halt;
...
```

与这些相似的代码构成了platform_init()函数的剩余部分。这里和平台相关的大多数代码需要和Linux内核通信。全局变量ppc_md提供钩子可以轻易地为PowerPC平台定制Linux内核，这个变量是machdep_calls结构，在.../arch/ppc/kernel/setup.c中定义。PowerPC相关的内核分支的许多地方都通过这个结构直接进行函数调用。例如，代码清单16-10中列出了.../arch/ppc/kernel/setup.c文件中的一部分内容，包括重启、关闭电源和停机函数。

代码清单16-10 通用的PowerPC机器函数

```
void machine_restart(char *cmd)
{
#ifdef CONFIG_NVRAM
    nvram_sync();
#endif
    ppc_md.restart(cmd);
}
void machine_power_off(void)
{
#ifdef CONFIG_NVRAM
```

```

        nvram_sync();
    #endif
        ppc_md.power_off();
    }

void machine_halt(void)
{
    #ifdef CONFIG_NVRAM
        nvram_sync();
    #endif
        ppc_md.halt();
}

```

这些函数通过ppc_md结构调用，加入了machine-或platform-specific变化。你可以看到其中的一些函数是和机器相关的，并且从mpc52xx_*函数改写后获得。这样的例子包括mpc52xx_restart和mpc52xx_map_io函数。另一些则指定了硬件平台。和平台相关的程序实例包括lite5200_map_irq和lite5200_setup_arch。

16.4 汇总

既然有了参考，我们就能为我们定制的开发板创建必要的条件和函数。我们复制Lite5200平台的文件作为工作基础，为特定的PowerPC平台进行修改。我们称新平台为PowerDNA，所执行移植的步骤如下：

- (1) 在...arch/ppc/Kconfig文件中增加一个新的配置选项；
- (2) 将lite5200.*复制到powerdna.*中，以此作为工作的基础；
- (3) 根据我们的平台需要，编辑新的powerdna.*文件；
- (4) 编辑.../arch/ppc/Makefile以包括powerdna.o；
- (5) 编译、载入以及调试。

第4章已经介绍了如何向Kconfig文件增加一个配置选项。移植新平台PowerDNA 的配置选项在代码清单16-11中给出。

代码清单16-11 PowerDNA的配置选项

```

config POWERDNA
    bool "United Electronics Industries PowerDNA"
    select PPC_MPC52xx
    help
        Support for the UEI PowerDNA board

```

Kconfig中的这个选项就加在LITE5200的下面，因为它们相关的。图16-4阐述了调用配置工具后的运行结果^①。

^① 为了节省篇幅，我们在图16-4中暂时删掉了LITE5200之前的机器类型。

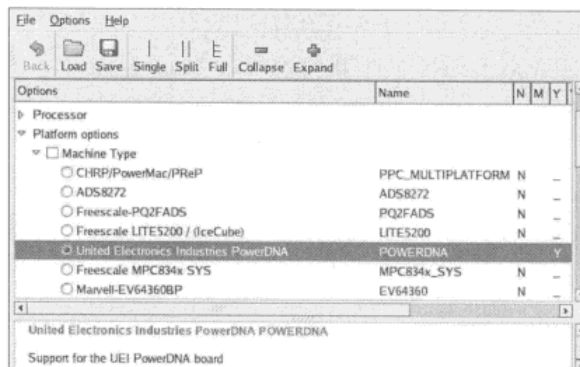


图16-4 PowerDNA的机器类型选项

注意，当用户选择了POWERDNA，那么系统将执行两个重要的动作：

(1) CONFIG_PPC_MPC52xx选项被自动选中。这一步是通过选择代码清单16-11中的关键字完成的。

(2) 定义一个新选项CONFIG_POWERDNA，编译时用到。

下一步是复制和我们平台最相近的文件，以此作为新平台相关文件的基础。我们认为Lite5200平台是合适的选择。将lite5200.c复制为powerdna.c，复制lite5200.h为powerdna.h。接下来介绍不同之处。根据硬件规范、原理图和硬件平台上的其他数据，编辑新的powerdna.*文件以适合新硬件使用。取得代码并编译，然后启动并调试新内核。这里没有捷径，经验是最重要的。移植是一项困难的工作，到那时现在至少你知道如何下手了。许多内核调试的提示和技巧在第14章中讲到。

为总结移植工作，代码清单16-12列出了PowerDNA开发板上运行Linux所需要增加或修改的文件列表。

代码清单16-12 PowerDNA 新的或修改的内核文件

```
linux-2.6.14/arch/ppc/configs/powerdna_defconfig
linux-2.6.14/arch/ppc/Kconfig
linux-2.6.14/arch/ppc/platforms/Makefile
linux-2.6.14/arch/ppc/platforms/powerdna.c
linux-2.6.14/arch/ppc/platforms/powerdna.h
linux-2.6.14/drivers/net/fec_mpc52xx/fec.c
linux-2.6.14/drivers/net/fec_mpc52xx/fec.h
linux-2.6.14/drivers/net/fec_mpc52xx/fec_phy.h
linux-2.6.14/include/asm-ppc/mpc52xx.h
```

第一个文件是默认的配置，该文件使得可以在默认配置的基础上快速配置内核。要使其生效可以这样调用：

```
$ make ARCH=ppc CROSS_COMPILE=<cross-prefix> powerdna_defconfig
```

我们已经讨论了Kconfig文件的改动。修改makefile文件不是太重要，其目的是根据CONFIG_POWERDNA配置选项增加对新内核的支持，改动是增加下面一行：

```
obj-$(CONFIG_POWERDNA) += powerdna.o
```

核心的改动在powerdna.[c|h]文件和FEC（快速以太网控制器）层。powerdna.c和lite5200.c有小小的不同，因为powerdna.c是从lite5200.c衍生过来的。有两处主要的问题需要改动。第一，PCI被禁用，因为PowerDNA的设计中没有使用PCI，因此需要一些调整。第二，在PowerDNA的设计中加入了一个不好操纵的以太网物理层芯片，它需要在硬件设置和FEC层进行微小的改动。这个工作是移植成功的关键点。补丁文件有1120行，但是大多数是默认的配置，只对开发人员有用，而且不是一定需要的。删掉以后，补丁文件剩下411行。

其他体系结构

我们详细介绍了一个给定的平台如何适合于内核，以及移植新平台的工具。本章提到和讨论的内容都来自内核的PowerPC体系结构分支。在细节上，其他体系结构会有许多不同点，但是概念是相似的。当你已经学会了如何驾驭一种体系结构，你就具备了学习其他体系结构的细节知识和手段。

16.5 小结

- 如果开发板使用Linux支持的处理器，那么将Linux移植到这块板子上相对简单，Linux代码基和硬盘平台的经验和知识是无可取代的。
- 使用内核已经支持的硬件平台的配置着手工作，能够为你的移植工作提供非常好的基础。
- 理解初始化代码流是轻松移植的关键。我们尽力不改变通用的内核代码，只修改那些平台本身需要的文件。本章很重要的一部分是讲与平台初始化相关的早期控制流。
- 进行下一步前，请加倍注意你的底层硬件平台初始化正确无误。举个例子说，如果发现你正在调试Linux Slab内存分配器中晦涩难懂的部分，我打赌你已经把某些内容和内存初始化弄混了。
- 本章的重点是Linux内核的PowerPC体系结构分支。掌握一种体系结构的细节为理解其余内核内容做了铺垫。

参考资源

Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture

MPCFPE32B/AD 12/2001 REV 2

Freescale Semiconductor, Inc.

MPC5200用户指南

MPC5200UG Rev 3 01/22005

Freescale Semiconductor, Inc.



本章内容

- 什么是实时
- 内核抢占
- 实时内核补丁
- 调试实时内核
- 小结

当Linux开始应用于Intel i386处理器时，没有人想到Linux会在服务器应用上获得成功。这份成功引导着Linux向各种不同体系结构的移植，从蜂窝电话到通信交换机，各行业的开发人员都使用Linux嵌入式系统。就在不久以前，如果你的应用有实时性要求，在你选择的操作系统中可能不包括Linux。现在情况发生了变化，因为大部分音频和多媒体应用程序上的实时Linux驱动程序都已经开发出来了。

本章首先简要介绍实时Linux的开发历程，然后着眼于实时程序设计中的可用设备以及如何使用这些设备。

17.1 什么是实时

如果你向5个人问起“实时（real time）”的含义，或许会得到5个不同的答案。一些回答甚至可能引用一些数字。下面就此进行讨论，先讨论一些情节，然后提出定义。一些需求可以被叫作软实时，而另外一些则被称为硬实时。

17.1.1 软实时

许多人认为软实时意味着操作具有时限（deadline），但是如果错过时限，操作质量可能会下降（但不会导致致命后果）。桌面工作站就是一个完美的软实时需求的例子。当你正在编辑文档时，希望按键结果立即在屏幕上显示；在你播放所喜爱的.mp3文件时，希望听到的是没有任何滴答声、爆裂声或者中断的高质量音频。

从术语上来讲，对于小于几十毫秒的延时，人们不会看到或者听到差别。当然，听众中的音乐家们会告诉你，即使比几十毫秒更短的延时都会影响音乐。如果没有所谓软实时事件的时限，

结果可能是不受大家欢迎的，但并不会是一场灾难。

17.1.2 硬实时

硬实时是以错过时限的结果为特征的。在一个硬实时系统中，如达不到时限，结果通常是灾难性的。当然，“灾难”是一个相对的术语。如果嵌入式设备控制喷气机引擎的燃料，若飞行员的输入或者修改操作标志错过了时限才响应，就会导致悲剧发生。

注意，时限的持续时间对实时的特征没有影响。原子钟上的滴答服务就是这样的例子。只要滴答在下个滴答之前的1秒窗口内被处理，数据仍然是合法的。但错过处理一个滴答也许会使我们的全球定位系统的误差达到1英尺甚至几英里！

鉴于此，我们为软实时和硬实时下一个通用的定义。对于软实时系统，如果错过时限，系统计算的价值或结果会有所减少；对于硬实时系统，如果某一次时限被错过，导致系统失败，或可能产生灾难性的后果。

17.1.3 Linux 调度

UNIX和Linux都设计成公平的进程调度算法。也就是说，调度器设法为所有需要CPU的进程最佳分配可用资源，并保证每个进程都能执行。这种特殊的设计宗旨与实时进程的需求相反。实时进程必须在它准备好之后就尽快运行。实时意味着有可预测性和反复性的潜在因素。

17.1.4 中断延迟

实时进程经常与一个物理事件相关，例如外围设备的中断到来。图17-1说明了Linux系统的中断延迟元素。测量延迟从接收到所需处理的中断开始，即图17-1中 t_0 之前的时间。过一段时间后，中断被接受并将控制权转交给中断服务程序（ISR），如图中的 t_1 。中断延迟几乎完全由最大化中断响应时间^①表示，这段时间指一个线程在禁止硬件中断的情况下的执行时间。

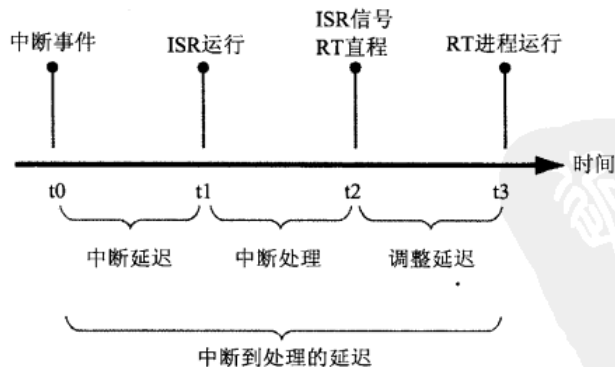


图17-1 中断延迟元素

① 我们忽略中断处理中上下文切换的时间，因为这段时间与中断处理时间相比是可以忽略的。

在实际中断服务程序中，好的设计经验是把处理时间减小到最小。实际上，执行的上下文受能力限制（例如，ISR无法调用阻塞函数，该函数也许处于睡眠态），因此，仅为硬件设备提供服务并将数据留给Linux下半部（bottom half）^①来处理，这么做是可取的，也称为软中断。

ISR/下半部已经完成处理后，通常会唤醒正在等待数据的用户空进程，如图17-1所示的t₂。在t₂之后，调度器选择运行实时进程并送给CPU，如图17-1所示的t₃。调度延迟主要受等待CPU的进程数和进程之间优先权的影响。在进程上设置实时属性，赋予普通Linux进程较高的优先权，假定等待CPU的实时进程有最高优先权，那么就允许选择运行下一个进程。只要最高优先权的实时进程想运行就可以运行（不受I/O阻拦）。你立刻就会看到如何设置这个属性。

17.2 内核抢占

早期的Linux1.x版本没有内核抢占。这意味着内核服务请求用户空进程时，除非其他进程受到阻拦（休眠）等待某事件（通常是I/O）或者完成内核请求，否则该进程就不能调度运行。进行内核抢占（preempt）^②意思是，内核中的一个进程运行时，其他进程可以抢占前一个还没有处理完的进程，如图17-2所示。

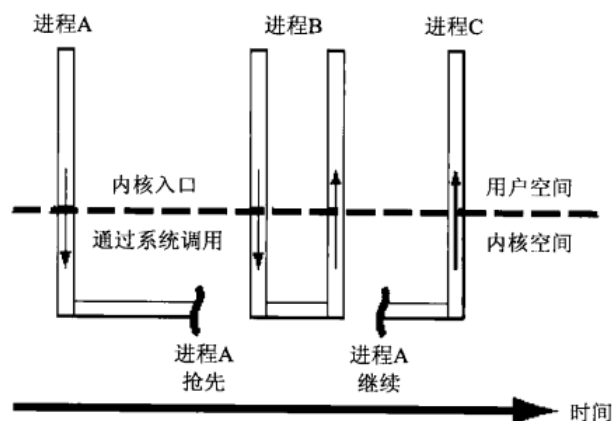


图17-2 内核抢占

图17-2中，进程A通过系统调用进入内核，或许这是对诸如控制台设备或文件设备的write()函数调用。当内核中的进程A正在运行时，具有较高优先权的进程B被中断唤醒。尽管进程A既没有受到阻拦也没有完成内核处理，但内核都会抢占进程A并将CPU分配给进程B。

17.2.1 抢占的缺陷

进行内核完全抢占的挑战是识别内核中所有不能被抢占的区域。内核中有一些关键部分不允

① Robert Love在他的《Linux内核开发》一书中详细解释了下半部处理，参见本章末的“参考资源”。

② 有趣的是，关于preemptable的正确拼写方法的辩论有许多！我采纳了Rick Lehrbaum的网上调查结果，www.linuxdevices.com/articles/AT5136316996.html。

许进行抢占。例如，假设图17-2中的进程A正在内核中执行文件系统操作。同时，代码也许需要向内核写一个数据结构以代表文件系统中的文件。为了保护这个数据结构不被破坏，该进程必须阻止其他进程访问共享的数据结构，代码清单17-1用C语言说明了这个概念。

代码清单17-1 锁定关键部分

```
...
preempt_disable();
...
/* Critical section */
update_shared_data();
...
preempt_enable();
...
```

如果我们没有用这种方式保护共有的数据，过程更新共有的数据结构可能在更新中途被抢占。如果其他过程试图更新相同的共享数据，则肯定会破坏数据的完整性。经典例子是两个进程直接操作同一变量并且要确定变量的取值。图17-3描述了这种情况。

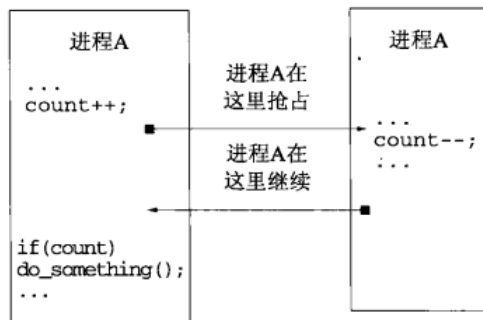


图17-3 共享数据并发时的错误

图17-3中，进程A在更新共享数据后但它赋值之前被中断。从设计角度讲，进程A不能检测到已被抢占。进程B在进程A再次运行之前修改了共享数据的值。可以看到，进程A将在进程B赋值基础上修改数据的值。如果这不是你要的结果，就必须在进程A输入数据、运行和赋变量计数值时使抢占无效。

17.2.2 抢占模型

第一个抢占内核的方法是在内核代码内部的关键位置放置检查记号，这样就知道在该位置抢占当前执行的线程是安全的。这些位置包括系统调用的入口和退出、一定内核锁的版本和中断处理程序的返回。这部分代码都与用来执行抢占操作的代码清单17-2类似。

代码清单17-2 为检查la Linux 2.4+抢占补丁进行的抢占

```
...
/*
 * This code is executed at strategic locations within
```



```

    * the Linux kernel where it is known to be safe to
    * preempt the current thread of execution
    */
    if (kernel_is_preemptable() && current->need_resched)
        preempt_schedule();
...

/*
 * This code is in ../kernel/sched.c and is invoked from
 * those strategic locations as above
 */
#ifdef CONFIG_PREEMPT
asmlinkage void preempt_schedule(void)
{
    while (current->need_resched) {
        ctx_sw_off();
        current->state |= TASK_PREEMPTED;
        schedule();
        current->state &= ~TASK_PREEMPTED;
        ctx_sw_on_no_preempt();
    }
}
#endif
...

```

代码清单17-2开始的代码片段（由实际代码中简化而来的）在前面提到的关键区域被调用，这些区域可以安全地进行抢占。代码清单17-2的第二个代码片段来自早期带有抢占补丁的Linux 2.4内核中的实际代码。非常有趣的是，直到所有抢占的请求都得到满足时，循环才会调用schedule()函数来切换上下文。

虽然这种方法可以使得Linux系统中的延时减少，但它并不是理想的方法。从事低延时处理的开发人员很快就会认识到“翻转逻辑”对于早期的抢占模型，我们有如下注意事项：

- Linux内核不可抢占；
 - 围绕内核在安全抢占的关键区域放置抢占标志；
 - 只有这些安全点可以抢占。
- 为了进一步地减少延迟，我们需要在抢占内核中进行如下处理：
- Linux内核处处可以抢占；
 - 仅在关键区域不能抢占。

这是内核开发人员比早期抢占内核补丁序列改进的地方。但是，这不是容易的任务。它包括浏览所有内核源代码，准确地分析什么数据应免于并发操作以及在哪些区域上使抢占无效。为此，解决方法是为内核提供测量延迟的设备，找到最长延迟代码路径并完善它们。最近Linux 2.6内核可能被配置为低延迟应用程序，因为其加入了“锁突破”（lock-breaking）方法。

17.2.3 SMP 内核

注意，创建高效多处理器体系结构的许多工作也有利于实时，这是很有趣的事情。SMP带来的挑战比单处理器复杂得多，因为需要保护额外的并发要素。在多处理器模式中，内核一次只会

执行一项任务，并发事件的保护只包含免受中断或者异常处理的保护。在SMP模式中，除了免受中断和异常处理的威胁外，内核还有可能执行多线程。SMP早在Linux 2.x内核中就已经获得了支持。大内核锁（BKL）可以用来保护从单处理器到SMP操作的转换并发操作。BKL是全局自旋锁，可以阻止内核中任何其他任务执行。在Robert Love的优秀著作《Linux内核开发（Novell出版社，2005）》中，他把BKL描述成“内核不喜欢的红发继子”。在描述BKL的特征时，Robert打趣地在属性清单中增加了“有害”项！

基于BKL的SMP内核早期的版本在调度上有着重大缺陷。它被发现其中一个CPU能在较长一段时间内无所事事。在做了许多工作以后，才产生了现在这样能够直接减少实时应用程序延时的高效SMP内核。用小粒度的锁保护实际共享数据的BKL可以大大减少抢占延迟。

17.2.4 抢占延迟源

实时系统必须能够在指定时间的上界内为实时任务服务。获得一致的低抢占延迟是实时系统的关键。对抢占延迟最大的两项贡献是中断上下文的处理和中断失效的关键区的处理。你已经学习了在开发板上减少关键区域的尺寸（还有，持续时间）上要进行很大的努力，剩下的就是中断上下文处理这个挑战了，这就是Linux 2.6实时补丁做的工作。

17.3 实时内核补丁

传统的kernel.org源码树并不支持硬实时。为了能够支持硬实时，必须打一个补丁。实时内核补丁是几种为减少Linux内核延时的努力成果的积累。实时内核补丁有许多贡献者，目前它由Ingo Molnar维护，你可以在<http://people.redhat.com/~mingo/realtime-preempt>找到。自从早期2.6内核发行至今，Linux 2.6内核已经有显著的改善。当第一次发布2.6版本时，2.4内核在软实时性能方面有极大的优势。自Linux 2.6.12，软实时性能（以毫秒计）在相当快的x86处理器上都能达到。为了获得更高的重复性性能需要实时补丁。

实时补丁为Linux内核增加了几个重要的特性。图17-4显示了应用实时补丁时的抢占模式的配置选项。

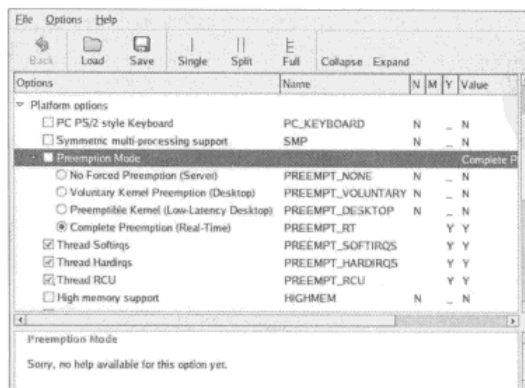


图17-4 带有实时补丁的抢占模式

实时补丁增加了4种抢占模式，称为PREEMPT_RT或者抢占实时。4种抢占模式如下所示。

- PREEMPT_NONE：非强制性抢占。一般来说整体延迟好，但是偶尔会有较长的延迟。最适合的应用是整体吞吐率都处于顶级的设计标准。
- PREEMPT_VOLUNTARY：减少延迟的第一个阶段。在内核关键区域放置额外的明确抢占点以便减少延迟。损失部分整体吞吐率换来低延迟。
- PREEMPT_DESKTOP：这种模式允许除了内核中的关键区域外都可以进行抢占。这种模式对软实时应用程序来说非常有效，例如音频和多媒体。整体吞吐率换来进一步地减少延迟。
- PREEMPT_RT：实时补丁增加的特性，包括用抢占互斥来代替自旋锁，这使得内核中不愿被抢占的区域受preempt_disable()保护。这种模式极大地消除了抖动(jitter)变化，并允许时间关键的实时应用具有较低的且可预测的延迟。

如果内核配置允许内核抢占，在内核命令行上增加下面的内核参数就能使启动时抢占无效：

```
preempt=0
```

17.3.1 实时的特性

CONFIG_PREEMPT_RT模式下有几个新的Linux内核特性是可用的。从图17-4中我们可以看到几种新的配置选项。这里会介绍这几个特性和实时Linux内核补丁的其他特性。

1. 旋锁转化为互斥

实时补丁会将系统中大多数自旋锁转化成互斥，这就允许以轻微损失吞吐率为代价来减少延迟。将自旋锁换成互斥的好处是后者可以进行抢占。如果进程A被锁住，具有较高优先权的进程B需要相同的锁，在有互斥的区域，进程A可以抢占进程B。

2. ISR 作为内核任务

选择CONFIG_PREEMPT_HARDIRQ模式，中断服务程序^①(ISR)强制运行进程上下文，这使得开发人员有控制ISR的优先权，因为它们变成了可调度的实体。同样，它们也成为可抢占的，以便允许较高优先权硬件中断先被处理。

这是个很重要的特点。一些硬件体系结构不强制执行中断优先权，这些硬件体系结构也许不需要强制执行优先权就与你所指定的实时设计目标一致。使用CONFIG_PREEMPT_HARDIRQ模式，在IRQ运行之前，你可以自由定义优先权。

若在/proc文件系统运行时或启动时在内核命令行中输入一个参数，ISR转换的线程是无效的。除非由你指定，否则开启这个配置，ISR线程默认情况下是启用的。为了使ISR线程在运行时无效，可以在根目录下输入下面的命令：

```
# echo '0' >/proc/sys/kernel/hardirq_preemption
```

为了确认该设置，可以将它按下述方式显示：

```
# cat /proc/sys/kernel/hardirq_preemption
1
```

^① 也称为 HARDIRQ。

为了使ISR线程在启动时无效，在内核命令行上添加下面参数：

```
hardirq-preempt=0
```

3. 抢占软中断

CONFIG_PREEMPT_SOFTIRQ模式通过在内核软中断后台程序(ksoftirqd)的上下文中运行软中断来减少延迟。ksoftirqd是一项适当的Linux任务(进程)。同样地，它能与其他任务一起被赋予优先权和调度。如果内核被配置为实时的，并开启了CONFIG_PREEMPT_SOFTIRQ模式，ksoftirqd内核任务就会被赋予实时优先权来处理软中断进程^①。代码清单17-3显示了从最新Linux内核中摘取的负责这项任务的代码，可以在.../kernel/softirq.c下找到。

代码清单17-3 赋予ksoftirq实时状态

```
static int ksoftirqd(void * __bind_cpu)
{
    struct sched_param param = { .sched_priority = 24 };

    printk("ksoftirqd started up.\n");

#ifdef CONFIG_PREEMPT_SOFTIRQS
    printk("softirq RT prio: %d.\n", param.sched_priority);
    sys_sched_setscheduler(current->pid, SCHED_FIFO, &param);
#else
    set_user_nice(current, -10);
#endif
    ...
}
```

这里我们看到的是如果内核配置CONFIG_PREEMPT_SOFTIRQS模式有效，那么在将内核函数sys_sched_setscheduler()的实时优先权设置为24时，ksoftirqd内核任务就会被提升为一项实时任务(SCHED_FIFO)。

软中断线程SoftIRQ可以在运行/proc文件系统过程中禁用，也可以通过在命令行上输入参数使其在启动时无效。当配置开启该功能时，除非你具体指定，否则软中断线程默认情况下是有效的。为了使软中断线程在运行时无效，在根目录下输入下面的命令：

```
# echo '0' >/proc/sys/kernel/softirq_preemption
```

要验证该设置，可以将其进行如下的显示：

```
# cat /proc/sys/kernel/softirq_preemption
1
```

为了使软中断线程在启动时禁用，可以在内核命令行上输入下面的参数：

```
softirq-preempt=0
```

4. 抢占RCU

RCU(Read-Copy-Update)^②是Linux内核中的一种特殊的同步原语，它是为会被频繁地读取

① 要了解有关软中断更多的知识，请参考本章末给出的参考资料《Linux内核开发》一书。

② 关于RCU的更深入讨论请访问www.rdrop.com/users/paulmck/RCU/。

但很少更新的数据而设计的。你可以把RCU当作优化的读锁。实时补丁增加了CONFIG_PREEMPT_RCU模式，通过抢占一定的RCU区域来改善延迟。

17.3.2 O(1) 调度器

从Linux 2.5版本开始就有了O(1)调度器。这里之所以提到是因为它是实时解决方法的关键因素。O(1)调度器比以前的Linux调度器改善了许多，它可以更好地规划带有多进程的系统，并有助于产生较低的总延迟。

你很疑惑，O(1)不是为一阶系统设计的数学名称吗？在本文中，它的意思是做出一个调度决定，不是根据进程的数量，而是根据给定的运行队列。旧版的Linux调度器没有这个特点，它的性能会随着进程的数量而下降。

17.3.3 创建实时进程

可以通过设置进程属性将其指定为实时的，这个进程会将调度器作为调度算法的一部分。代码清单17-4显示了通用实现方法。

代码清单17-4 创建实时进程

```
#include <sched.h>

#define MY_RT_PRIORITY MAX_USER_RT_PRIO /* Highest possible */

int main(int argc, char **argv)
{
    ...
    int rc, old_scheduler_policy;
    struct sched_param my_params;
    ...

    /* Passing zero specifies caller's (our) policy */
    old_scheduler_policy = sched_getscheduler(0);

    my_params.sched_priority = MY_RT_PRIORITY;
    /* Passing zero specifies callers (our) pid */
    rc = sched_setscheduler(0, SCHED_RR, &my_params);
    if ( rc == -1 )
        handle_error();
    ...
}
```

这段代码调用sched_setscheduler()函数完成了两件事。它将调度方法变成SCHED_RR，并将其在系统上的优先权提高到最可能的大。Linux支持如下3种调度方法。

- SCHED_OTHER: 通用的Linux进程，公平调度。
- SCHED_RR: 带有时间段的实时进程，也就是说，如果没有阻拦，它可以在调度器给定的时间段中运行。
- SCHED_FIFO: 直到它阻塞或者明确放弃处理器，或者直到另一个较高优先权的SCHED_FIFO进程运行，实时进程运行才停止运行。

`sched_setscheduler`的手册提供了关于这三种不同的调度方法的说明。

17.3.4 临界区管理

编写内核代码（如客户设备驱动程序时），你会遇到必须避免并发访问的数据结构。最简单的保护关键数据的方法是禁用周遭的抢占，尽可能保持关键路径短以便保证系统的延迟最低。代码清单17-5演示了一个实例。

代码清单17-5 在内核代码中保护关键区域

```
...
/*
 * Declare and initialize a global lock for your
 * critical data
 */
DEFINE_SPINLOCK(my_lock);
...

int operate_on_critical_data()
{
    ...
    spin_lock(&my_lock);
    ...
    /* Update critical/shared data */
    ...
    spin_unlock(&my_lock);
    ...
}
```

当一个任务成功获取自旋锁时，抢占是无效的，并且允许自旋锁进入关键区域。直到`spin_unlock`操作运行后，才能进行任务切换。`spin_lock()`实际上是一个有几种形式的宏，其形式主要由内核配置决定。它们是在顶层文件（独立体系结构定义）`.../include/linux/spinlock.h`中定义的。当用实时补丁补充内核时，这些自旋锁转换成互斥体，以便允许较高优先权的进程一拥有自旋锁就进行抢占。

由于实时补丁对于设备驱动程序和内核开发人员很大程度上都是透明的，所以可以使用熟悉的结构来保护关键区域，正如代码清单17-5描述的那样。这是在实时应用程序上打实时补丁的主要优点，它会保留熟悉的锁符号和中断服务程序。

正如代码清单17-5中那样，使用宏`DEFINE_SPINLOCK`为以后的版本保留兼容性。这些宏在`.../include/linux/spinlock_types.h`文件中定义。

17.4 调试实时内核

有几个配置选项促进实时内核补丁的调试和性能分析，下面详细分析这些配置。

17.4.1 软锁检测

为了启用软锁检测功能，先要启用内核配置中的`CONFIG_DETECT_SOFTLOCKUP`选项。这项功

能允许长时间运行在没有任何上下文切换的内核模式下的检测。这项功能存在于非实时内核中，但是它对检测高延迟或软死锁环境非常有用。为了使用这项功能，只需开启这项功能，并监视控制台或系统日志上的任何报告。生成的报告与下面类似：

```
BUG: soft lockup detected on CPU0
```

这个消息发给内核时，通常会伴有backtrace命令和其他信息（例如进程名和PID）。它看起来与用处理器的寄存器完成的内核oops消息类似。参见/kernel/softlockup.c获取详细信息。这条信息有助于获得产生锁条件原因。

17.4.2 抢占调试

为了开启抢占调试功能，在内核配置中要启动CONFIG_DEBUG_PREEMPT选项。这个调试参数支持检测抢占语义的非安全性使用，例如当在无效的上下文中时，抢占计数值下溢并试图休眠。为了使用这个功能，只需开启这个参数，并监视任何控制台或系统日志中的报告。这里只是给出当抢占调试开启后的一个实例报告。

```
BUG: <me> <mypid>, possible wake_up race on <proc> <pid>
BUG: lock recursion deadlock detected! <more info>
BUG: nonzero lock count <n> at exit time?
```

可能会有更多的消息，这些只是一些可以被检测到的问题的例子。这些消息可以帮助你在使用实时内核参数时避免死锁和其他错误或危险的程序设计语义。有关参数发出的消息和环境的更详细的信息，请浏览Linux内核源文件.../kernel/rt-debug.c。

17.4.3 调试唤醒时间

为了开启唤醒时间功能，在内核配置中选中CONFIG_WAKEUP_TIMING选项。当在CPU上调度一个高优先权的进程时，这个调试选项就启动测量唤醒该进程的时间。这个选项很容易使用。当配置后，测量功能无效。为了开启测量功能，使用超级用户执行下面语句：

```
# echo '0' >/proc/sys/kernel/preempt_max_latency
```

当/proc系统中的这个文件被设置成0时，每个连续最大唤醒时间的结果都写入到这个文件中。为了读取当前最大值，只需显示该值：

```
# cat /proc/sys/kernel/preempt_max_latency
84
```

只要延迟测量模式在内核配置中启用了，preempt_max_latency都将用最大延迟值更新。它不能禁用。将0写入到/proc文件的变量中，可以将最大值重新设置成0来重启累加测量。

17.4.4 唤醒延迟历史

为了启动唤醒延迟历史，启动选项CONFIG_WAKEUP_LATENCY_HIST的同时也启动CONFIG_WAKEUP_TIMING选项。这个选项将由CONFIG_WAKEUP_TIMING选项启动的唤醒时间测量转储到一个文件中，方便以后的分析。很快我们会在检查中断历史时，举一个这样文件和内容的例子。

- CRITICAL_PREEMPT_TIMING: 测量花费在关键的抢占无效区的时间。
- PREEMPT_OFF_HIST: 与WAKEUP_LATENCY_HIST选项类似, 将抢占时间测量收集到一份二进制文件中, 以便以后分析。

17.4.5 中断响应时间

为了启用最大中断响应时间的测量, 在内核配置中要开启CRITICAL_IRQSOFF_TIMING选项。这个选项测量用在irqs无效时关键区的时间。这个选项与唤醒延迟时间工作方式相同。为了启用这种测量方法, 应以超级用户身份执行下面命令:

```
# echo '0' >/proc/sys/kernel/preempt_max_latency
```

当/proc系统下的这个文件被设置成0, 各个连续最大中断响应时间的结果都被写到这个文件中。为了读取当前的最大值, 只需显示该值:

```
# cat /proc/sys/kernel/preempt_max_latency
97
```

你会注意到启动唤醒延迟和中断延迟的测量时, 都使用了同一个/proc系统下的文件。当然, 这就意味着一次只能配置一个测量, 否则结果可能是无效的。因为这些测量增加了许多运行时间, 所以同时启动这些测量是非常不明智的。

17.4.6 中断响应历史

INTERRUPT_OFF_HIST启动提供的功能与WAKEUP_LATENCY_HIST提供的类似。这个选项为了以后分析方便, 将中断延迟时间测量值收集到一个文件中。数据以柱状图格式存储, 柱的范围从0到10 000多毫秒。在前面的例子中, 我们看到那次特殊采样的最大延迟是97毫秒。因此, 可以得出结论, 柱状图形式的延迟数据不包含任何超过97毫秒二进制文件的有用信息。

读取/proc系统下的文件可以获取历史数据。输出被重新导入到一个规则文件中来分析, 或者以下面方式出现:

```
# cat /proc/latency_hist/interrupt_off_latency/CPU0 > hist_data.txt
```

代码清单17-6显示了前10行历史数据。

代码清单17-6 中断延迟历史 (头)

```
$ cat /proc/latency_hist/interrupt_off_latency/CPU0 | head
#Minimum latency: 0 microseconds.
#Average latency: 1 microseconds.
#Maximum latency: 97 microseconds.
#Total samples: 60097595
#There are 0 samples greater or equal than 10240 microseconds
#usecs      samples
  0          13475417
  1          38914907
  2          2714349
  3          442308
...
```


从代码清单17-6中可以看到最大值和最小值、所有值的平均值和采样总数。本例中，我们只采集了60 000 000样值。柱状图数据后面跟有总数并包含大约10 000二进制数据。利用gnuplot可以很容易绘制这些数据，如图17-5所示。

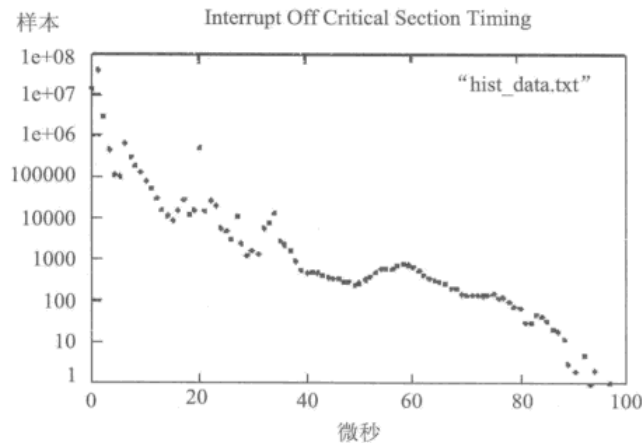


图17-5 中断响应延迟数据

17.4.7 延迟跟踪

LATENCY_TRACE配置选项启动了与最新的最大延迟测量相关的内核跟踪过程。它也可以通过/proc文件系统启动。延迟跟踪可以帮助你隔离最长的延迟代码路径。对于每次新的最大延迟测量，都会产生一个相关的跟踪来方便跟踪有关最大延迟的代码路径。

代码清单17-7复制了跟踪测量最长78ms的例子。与使用其他测量工具一样，将0写入到/proc/sys/kernel/preempt_max_latency文件中来启动测量。

代码清单17-7 中断响应最大延迟跟踪

```
$ cat /proc/latency_trace
preemption latency trace v1.1.5 on 2.6.14-rt-intoff-tim_trace
-----
latency: 78 us, #50/50, CPU#0 | (M:rt VP:0, KP:0, SP:1 HP:1)
-----
| task: softirq-timer/0-3 (uid:0 nice:0 policy:1 rt_prio:1)
-----

-----=> CPU#
/ -----=> irqsoft
| / -----=> need-resched
|| / -----=> hardirq/softirq
||| / -----=> preempt-depth
|||| /
|||||
cmd      pid      delay
\      /      \      /
cat-6637 0D... 1us : common_interrupt ((0))
```

```

cat-6637 0D.h. 2us : do_IRQ (c013d91c 0 0)
cat-6637 0D.h1 3us+: mask_and_ack_8259A (__do_IRQ)
cat-6637 0D.h1 10us : redirect_hardirq (__do_IRQ)
cat-6637 0D.h. 12us : handle_IRQ_event (__do_IRQ)
cat-6637 0D.h. 13us : timer_interrupt (handle_IRQ_event)
cat-6637 0D.h. 15us : handle_tick_update (timer_interrupt)
cat-6637 0D.h1 16us : do_timer (handle_tick_update)
... <we're in the timer interrupt function>
cat-6637 0D.h. 22us : run_local_timers (update_process_times)
cat-6637 0D.h. 22us : raise_softirq (run_local_timers)
cat-6637 0D.h. 23us : wakeup_softirqd (raise_softirq)
... <softirq work pending - need to preempt is signaled>
cat-6637 0Dnh. 34us : wake_up_process (wakeup_softirqd)
cat-6637 0Dnh. 35us+: rcu_pending (update_process_times)
cat-6637 0Dnh. 39us : scheduler_tick (update_process_times)
cat-6637 0Dnh. 39us : sched_clock (scheduler_tick)
cat-6637 0Dnh1 41us : task_timeslice (scheduler_tick)
cat-6637 0Dnh. 42us+: preempt_schedule (scheduler_tick)
cat-6637 0Dnh1 45us : note_interrupt (__do_IRQ)
cat-6637 0Dnh1 45us : enable_8259A_irq (__do_IRQ)
cat-6637 0Dnh1 47us : preempt_schedule (enable_8259A_irq)
cat-6637 0Dnh. 48us : preempt_schedule (__do_IRQ)
cat-6637 0Dnh. 48us : irq_exit (do_IRQ)
cat-6637 0Dn.. 49us : preempt_schedule_irq (need_resched)
cat-6637 0Dn.. 50us : __schedule (preempt_schedule_irq)
... <here is the context switch to softirqd-timer thread>
<...>-3 0D..2 74us+: __switch_to (__schedule)
<...>-3 0D..2 76us : __schedule <cat-6637> (74 62)
<...>-3 0D..2 77us : __schedule (schedule)
<...>-3 0D..2 78us : trace_irqs_on (__schedule)
... <output truncated here for brevity>

```

为了清晰起见，我们整理了这份代码清单，但还是可以很明显地看到跟踪的关键元素。这次跟踪由定时器的中断产生。在hardirq线程中，除了为以后的softirq上下文排列了一些作业，其他没有做什么。这点可以在23毫秒处通过wakeup_softirqd()函数来查看，这是一个典型的中断处理程序。该函数触发need_resched标志，正如跟踪的秒区第3列中由n显示的那样。在49毫秒处，在定时器softirq中进行一些处理之后，调度器请求抢占。在74毫秒处，控制权会转交给以PID3形式运行在特殊内核中的实际softirq定时器0线程（进程名被裁减到合适宽度并用<...>显示）。

代码清单17-7中大多数字段有明显的意义。irqs-off字段包含一个D，这个字段是中断响应的代码字段。因为延迟跟踪是中断响应跟踪，我们看着它来指导全部跟踪。need_resched字段反映了内核need_resched标志的状态。n说明调度器应该尽快运行，句点(.)的意思是这个标志没有激活。hardirq/softirq字段说明在hardirq上下文中用h执行线程，在softirq上下文中用s执行线程。preempt-depth字段给出了内核preempt_count变量的值，preempt_count是内核中锁的嵌套层的指示器。只有当变量是0时，才允许抢占。

17.4.8 调试死锁环境

内核配置DEBUG_DEADLOCKS选项允许对死锁环境的进行检测和报告，这些死锁与内核中的信

号量和自旋锁有关。当选项启用时，潜在的死锁环境就会以与下面类似的方式报告：

```
=====
[ BUG: lock recursion deadlock detected! |
-----
...
```

在通知死锁检测的标题后显示了许多信息，包括锁的描述符、锁名（如果可用）、锁的文件和文件名称（如果可用）、锁主（就是当前拥有锁的人），等等。直接确定违规进程是可能的，当然，修补这个进程也许不那么简单了！

17.4.9 锁模式的运行时控制权

DEBUG_RT_LOCKING_MODE选项允许将实时互斥的运行时控制权转回为非抢占模式，这么做可以有效地把实时内核（自旋锁与互斥）的运行转回给基于自旋锁的内核。与这里谈论的其他配置一样，这个工具可以被认为是一个开发工具，目的是只在开发环境中使用该工具。

立即启动所有调试模式是没有意义的。也许你可以想象，大多数调试模式使内核增大了容量并增加了许多处理进程。这些都意味着调试模式作为开发辅助工具使用时，这些模式应该对生产代码是无效的。

17.5 小结

- Linux越来越被广泛地用于需要实时性能的系统中，包括多媒体应用程序和机器人、工业和自动控制器。
- 实时系统是以时限为特征的。当系统一次错过时限，产生的结果只是让客户感觉到不便或降低了客户体验，我们将该系统称为软实时系统。反之，硬实时系统是指一次错过时限将导致系统失败。
- 内核抢占是Linux内核最重要的特点，它说明了系统广义延迟。
- 新Linux内核支持几种抢占模式，从非抢占模式到全实时抢占模式。
- 实时补丁为Linux内核增加了几种关键特性，可以产生低延迟。
- 实时补丁包含几种重要的测量工具，可以帮助调试并表征实时Linux的实现。

参考资源

Linux 内核开发，第2版
Robert Love
Novell Press, 2005

附录 A 可配置的 U-Boot 命令

(续)

命 令 集	命 令
CFG_CMD_BEDBUG	包含 BedBug调试器
CFG_CMD_FDC	支持软驱
CFG_CMD_SCSI	支持SCSI
CFG_CMD_AUTOSCRIP	支持自动脚本
CFG_CMD_MII	支持MII
CFG_CMD_SETGETDCR	支持DCR
CFG_CMD_BSP	板极相关的函数
CFG_CMD_ELF	ELF (VxWorks) 载入/启动命令
CFG_CMD_MISC	其他功能, 例如休眠
CFG_CMD_USB	支持USB
CFG_CMD_DOC	支持Disk-on-chip
CFG_CMD_JFFS2	支持JFFS2
CFG_CMD_DTT	数字调温器
CFG_CMD_SDRAM	SDRAM信息打印
CFG_CMD_DIAG	诊断
CFG_CMD_FPGA	支持FPGA配置
CFG_CMD_HWFLOW	RTS/CTS硬件流控制
CFG_CMD_SAVES	保存S记录转储
CFG_CMD_SPI	SPI 实用工具
CFG_CMD_FDOS	支持DOS
CFG_CMD_VFD	支持VFD (TRAB)
CFG_CMD_NAND	支持NAND
CFG_CMD_BMP	支持BMP
CFG_CMD_PORTIO	I/O口
CFG_CMD_PING	支持Ping
CFG_CMD_MMC	支持MMC
CFG_CMD_FAT	支持FAT
CFG_CMD_IMLS	列出找到的映像
CFG_CMD_ITEST	整数 (和字符串) 测试
CFG_CMD_NFS	支持NFS
CFG_CMD_REISER	支持Reiserfs
CFG_CMD_CDP	Cisco (思科) 发现协议
CFG_CMD_XIMG	载入multi-image
CFG_CMD_UNIVERSE	支持Tundra Universe
CFG_CMD_EXT2	支持EXT2
CFG_CMD_SNTP	支持SNTP
CFG_CMD_DISPLAY	支持显示

附录 A

可配置的U-Boot命令

U-Boot有60多个可配置的命令，表A-1总结了U-Boot中的这些命令。此外，还有很多非标准命令，其中一些和硬件相关或还在实验阶段。最完整的命令列表，请参考源代码。这些命令在U-Boot源码中的.../include/cmd_confdefs.h头文件中定义。

表A-1 可配置的U-Boot命令

命 令 集	命 令
CFG_CMD_BDI	bdinfo
CFG_CMD_LOADS	loads
CFG_CMD_LOADB	loadb
CFG_CMD_IMI	iminfo
CFG_CMD_CACHE	icache、dcache
CFG_CMD_FLASH	flinfo、erase、protect
CFG_CMD_MEMORY	md、mm、nm、mw、cp、cmp、crc、base、loop、mtest
CFG_CMD_NET	bootp、tftpboot、rarpboot
CFG_CMD_ENV	saveenv
CFG_CMD_KGDB	kgdb
CFG_CMD_PCMCIA	支持PCMCIA
CFG_CMD_IDE	支持IDE硬盘
CFG_CMD_PCI	pciinfo
CFG_CMD_IRQ	irqinfo
CFG_CMD_BOOTD	bootd
CFG_CMD_CONSOLE	coninfo
CFG_CMD_EEPROM	支持EEPROM读写
CFG_CMD_ASKENV	获得环境变量
CFG_CMD_RUN	在环境变量中运行命令
CFG_CMD_ECHO	echo参数
CFG_CMD_I2C	支持I2C串行总线
CFG_CMD_REGINFO	注册转储
CFG_CMD_IMMAP	支持IMMR 转储
CFG_CMD_DATE	支持RTC、日期/时间等等
CFG_CMD_DHCP	支持DHCP

BusyBox有很多有用的命令。下面是近期BusyBox版本中的命令。

addgroup	在系统中增加一个组
adduser	在系统中增加一个用户
adjtimex	读取和有选择地设置系统中和时间相关的参数
ar	从一个ar归档中提取或列出文件
arping	使用ARP请求/回复ping主机
ash	ash shell (命令解释程序)
awk	模式扫描和处理语言
basename	去除目录路径和后缀的文件
bunzip2	解压缩文件 (或如果没有指定输入文件, 为标准输入)
bzcat	解压缩到标准输出
cal	显示日历
cat	连接多个文件并打印到标准输出
chgrp	改变一个文件所属的组
chmod	改变文件访问权限
chown	改变文件所有权
chroot	用新的超级用户运行命令
chvt	改变前台虚拟终端到/dev/ttyN
clear	清屏
cmp	比较文件差异
cp	复制文件
cpio	从cpio归档中提取或列出文件
cron	cron的BusyBox版本
crontab	管理crontab 控制文件
cut	打印输入文件中选择的字段到标准输出
date	显示或设置系统时间
dc	小型RPN计算器
dd	根据选项复制, 转换和格式化文件
deallocvt	重新分配不用的虚拟终端/dev/ttyN



delgroup	从系统中删除一个组
deluser	从系统中删除一个用户
devfsd	管理devfs 权限和老设备名符号链接的守护进程, 现在已经过时
df	打印使用的文件系统空间和可用的空间
dirname	去除文件名中非目录后缀
dmesg	打印或控制内核环缓冲区
dos2unix	将文件从DOS格式转换到UNIX格式
dpkg	安装、删除和管理Debian软件包的工具
dpkg-deb	管理Debian软件包 (debs)
du	显示每一个文件/目录占用的磁盘空间
dumpkmap	打印一个二进制键盘转换表到标准输出
dumpleases	显示udhcpd准许的DHCP 租借物
echo	打印指定的ARG到标准输出
env	打印当前的环境变量或在设置后运行程序
expr	打印表达式的值到标准输出
false	返回FALSE (1)的退出码
fbset	显示和修改帧缓冲 (frame buffer) 设置
fdflush	强制软盘驱动器检测磁盘的变化
fdformat	低级格式化一张软盘
fdisk	改变分区表
find	在目录中查找文件
fold	限制文件列宽
free	显示系统中剩余和使用的内存数量
freeramdisk	释放指定的ramdisk使用的所有内存
fsckminix	为MINIX文件系统执行一致性检查
ftpget	通过FTP获得远处的一个文件
ftpput	通过FTP将本地文件上传到远程
getopt	解析命令选项
getty	打开一个终端, 提示输入用户名, 然后调用/bin/login
grep	搜索含有给定字符串或模式的一个或多个文件
gunzip	解压缩文件 (即标准输入)
gzip	最大程度压缩文件
halt	系统停机
hdparm	获得/设置硬盘参数
head	打印一个文件的前10行到标准输出
hexdump	转储文件, 格式可以是用户指定的二进制、八进制、十六进制、字符或十进制
hostid	打印唯一的32位机器标识符
hostname	获得或设置主机名
httpd	侦听到来的http服务请求
hwclock	查询和设置硬件时钟 (RTC)

id	打印当前用户信息
ifconfig	配置网络接口
ifdown	取消网络接口的配置
ifup	配置网络接口
inetd	侦听网络连接和发出的程序
init	init的BusyBox 版本
insmod	将指定的内核模块加载到内核中
install	复制文件，并设置属性
ip	TCP/IP 配置程序
ipaddr	处理接口地址
ipcalc	根据IP地址计算IP网络设置
iplink	处理接口设置
iproute	显示/设置路由
iptunnel	BusyBox中的iptunnel实用程序
kill	给指定的进程发送一个信号（默认信号是SIGTERM）
killall	给指定的进程发送一个信号（默认信号是SIGTERM）
klogd	内核记录器
lash	BusyBox LAme Shell（命令解析器）
last	显示最后登录系统的用户列表
length	打印指定字符串的长度
ln	为指定目标创建一个名为LINK_NAME或DIRECTORY的链接
loadfont	从标准输入载入终端字体
loadkmap	从标准输入加载键盘转换表
logger	向系统日志写入消息
login	在系统上开始一个新会话
logname	打印当前用户的用户名
logread	显示来自syslogd的消息
losetup	把LOOPDEVICE 和文件关联到一起
ls	列出目录
lsmod	列出载入到内核中的模块
makedevs	创建几个块或字符设备文件
md5sum	打印或检查MD5校验和
mesg	控制是否允许他人传信息到自己的终端
mkdir	创建一个目录项
mkfifo	创建一个命名管道（和mknod p一样）
mkfsminix	创建一个MINIX文件系统
mknod	创建一个特定文件（块、字符或管道）
mkswap	准备一个磁盘分区作为交换分区
mktemp	创建一个临时文件，其名基于TEMPLATE
modprobe	用于高级模块载入和卸载



more	一次一页地显示文件
mount	挂载文件系统
mt	控制磁带驱动器的操作
mv	重命名或转移文件
nameif	在关闭状态时, 重命名网络接口
nc	设置路由器
netstat	显示 Linux 网络信息
nslookup	查询命名服务器, 以获得主机IP地址
od	以八进制和其他格式转储文件
openvt	在一个新的虚拟终端开始命令
passwd	修改用户密码
patch	patch命令的BusyBox版本
pidof	获得进程的PID
ping	发送ICMP ECHO_REQUEST 数据包到网络主机
ping6	发送ICMP ECHO_REQUEST 数据包到网络主机
pivot_root	修改根文件系统
poweroff	关闭系统, 要求内核关闭电源
printf	根据用户格式, 格式化和打印参数
ps	报告进程的状态
pwd	打印当前工作目录的完整文件名
rdate	远程获得(或设置)系统日期和时间
readlink	显示符号链接的值
realpath	返回给定参数的绝对路径
reboot	系统重启动
renice	改变进程优先级
reset	将屏幕复位
rm	删除(解除链接)文件
rmdir	删除空目录
rmmod	卸载指定的内核模块
route	编辑内核的路由表
rpm	管理 RPM 包
rpm2cpio	从 RPM 软件包中提取cpio归档
run-parts	运行目录中的脚本
rx	使用xmodem协议接收文件
sed	流编辑器实现的Busybox版本
seq	以指定的步长输出一个数列
setkeycodes	将项装入内核的扫描码/键码映像表
shasum	打印或检查SHA1校验和
sleep	延时一段时间
sort	排列指定文件中的文本



start-stop-daemon	启动和停止设备的守护程序
strings	显示一个二进制文件中可打印的字符串
stty	显示和修改终端设置
su	改变用户ID或成为root
sulogin	单用户登录
swapoff	禁用虚拟内存交换
swapon	启用虚拟内存交换
sync	将内存缓冲区内的数据写入磁盘
sysctl	在启动时配置内核参数
syslogd	Linux系统和内核日志实用程序
tail	打印文件的最后10行
tar	创建、展开或列出tar文件
tee	读取标准输入的数据，并将其内容输出成标准输出
telnet	Telnet 客户端的BusyBox版本
telnetd	Telnet 服务器端的BusyBox版本
test	检查文件类型并进行比较
tftp	使用TFTP协议传送文件
time	计算程序使用的时间
top	实时提供处理器活动情况
touch	更新所给定文件的最后修改日期
tr	传送、压缩和删除字符
traceroute	追踪IP包跟踪的路由
true	返回退出码TRUE (0)
tty	打印连接到标准输入的终端的文件名
udhcp	DHCP客户端的BusyBox版本
udhcpd	DHCP服务器端的BusyBox版本
umount	卸载文件系统
uname	打印当前系统信息
uncompress	解压缩Z文件
uniq	去除文件中同样的行
unix2dos	把文件从UNIX格式转换为DOS格式
unzip	解压缩ZIP文件
uptime	显示自上次启动后的时间
usleep	暂停 n 毫秒
uudecode	将使用uuencode编码后的文件还原
uencode	编码并压缩文件
vconfig	创建和删除虚拟以太网设备
vi	vi编辑器的BusyBox版本
vlock	给虚拟终端上锁，使用密码解锁
watch	周期性地执行程序



watchdog	周期性地写看门狗设备
wc	打印文件的行、字和字节数
wget	通过HTTP或FTP接收文件
which	在当前的路径中定位一个命令
who	打印当前系统的所有用户名和相关信息
whoami	打印当前用户的用户名
xargs	从标准输出产生执行命令行
yes	用指定的字符串或y再次输出一行
zcat	解压缩到标准输出



SDRAM接口的注意事项

本附录内容

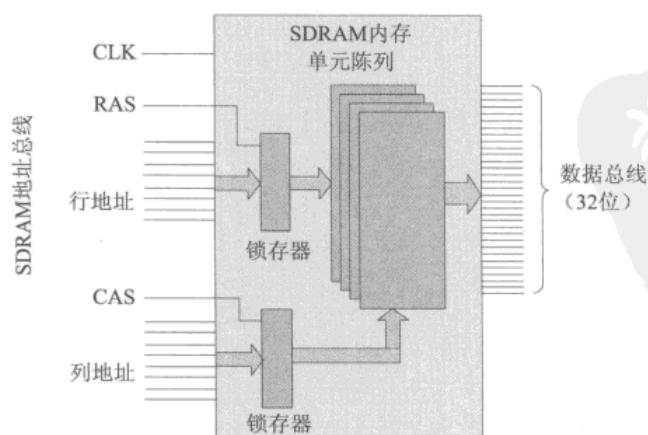
- SDRAM基础
- 时钟
- SDRAM设置
- 小结

乍一看，为SDRAM控制器编程似乎是一项艰巨的任务。实际上，许多同步动态随机存储器（DRAM）技术已经被开发了。随着性能和密度的无止境需求，许多针对不同体系结构和操作模式的程序已经开发出来。

有关SDRAM接口的注意事项，我们分析AMCC PowerPC 405GP处理器。当我们探讨与SDRAM接口有关的问题时，你可能需要参考用户手册。“参考资源”中引用了这篇文档。

C.1 SDRAM 基础

为了理解SDRAM的设置，必须先理解SDRAM设备是怎么运行的。不需要深入硬件设计的细节，一个SDRAM设备组织为单元阵列，一些地址位表示行地址，一些地址位表示列地址。如图C-1所示。



图C-1 简化的SDRAM 模块图

矩阵内部的电路非常复杂。下面举一个简单的读操作例子。在行地址上放置行地址，在列地址上放置列地址，这样就获得了一个给定的内存区域。一段时间以后，处理器会把存储在该位置的数据放在数据总线上。

处理器向SDRAM地址总线输出一个行地址，并给出它的行地址选择（RAS，Row Address Select）信号。一个暂短的延时过后，允许SDRAM电路捕捉行地址。而处理器输出一个列地址，给出它的列地址选择（CAS，Column Address Select）信号。SDRAM控制器把实际的物理内存地址翻译为行、列地址。很多SDRAM控制器可以使用行和列的宽度进行配置，PPC405GP就是其中一个例子。一会儿你将看到，设置PPC405GP的SDRAM控制器时，必须对它进行配置。

虽然这个例子非常简单，但是概念是一样的。例如，如果SDRAM控制器突发4个内存地址，那么它输出一个RAS和CAS周期，然后内部的SDRAM电路自动增加后面三个突发读的列地址，这就省去了处理器要发送4次单独的CAS周期。这只是一个性能优化的例子。理解这部分内存的最好方法是掌握实际存储器芯片的细节。在“参考资源”中有一篇非常好的数据手册的例子。

SDRAM 刷新

SDRAM由单个晶体管和电容组成。晶体管负责提供电荷，而电容的工作是保持（存储）各个单元的值。电容可以在一段周期内保持数值，其原因已经超出了本书的范围。动态存储的基本概念之一是，每一个单元必须定期充电才能保持里面的值。这个过程被称为SDRAM刷新。

一个刷新周期是一个特殊的内存周期，它不对内存进行任何的读写操作，只是执行所需要的刷新周期。SDRAM控制器的一个主要职责就是保证刷新周期及时地发生，以满足芯片的需要。

芯片厂商指定了最小的刷新时间间隔，保证这段间隔是设计者的工作。通常SDRAM控制器可以配置为直接刷新间隔。这里提到的PowerPC 405GP就有一个寄存器完成这项功能。我们很快就会看到。

C.2 时钟

术语“同步”是指一个SDRAM设备的数据读和写周期与来自处理器的时钟信号一致。SDR SDRAM（单一数据速率SDRAM）在每一次SDRAM时钟周期内进行读写。DDR SDRAM（双重数据速率SDRAM）在一个周期内读写两次，一次是时钟的上升沿，一次是下降沿。

现代处理器具有复杂的时钟子系统。很多处理器都有多个时钟比率，用在系统中的不同部分。一个典型的处理器使用相对低频的晶振源作为它的主时钟信号，处理器内部的锁相环产生CPU的主时钟（在比较处理器速度时，我们会讲时钟比率）。因为CPU通常运行得比内存子系统更快，所以处理器会产生一个主CPU时钟的1/N提供给SDRAM子系统。你需要根据你的CPU和SDRAM组合配置这个时钟比率。

处理器和内存子系统的时钟必须配置正确，否则SDRAM不会正常工作。处理器手册中包含了一段关于时钟设置和管理的内容，你必须参考这些内容，正确地设置你的特殊设计。

AMCC 405GP是这样典型的。它采用单晶振产生时钟输入源，产生几个子系统需要的内部和外部时钟。它产生的时钟用于CPU、PCI接口、板上外围总线（Onboard Peripheral Bus，OPB）、

处理器局部总线（Processor Local Bus, PLB）、内存时钟（MemClk）和几个外围设备（如定时器和UART功能块）。一个典型的配置可能和表C-1相似。

表C-1 PPC405GP的典型时钟配置

时 钟	比 率	注 释
晶振参考	33 MHz	处理器提供的基本参考值
CPU 时钟	133 MHz	根据处理器的内部PLL衍生而来，由硬件管脚侦测（strapping）和寄存器设置控制
PLB 时钟	66 MHz	根据处理器的时钟、硬件管脚侦测和寄存器设置衍生而来。用于内部处理器局部总线上的高速模块间的数据交换
OPB 时钟	66 MHz	根据PLB的时钟和寄存器设置衍生而来。用于不需要高速连接的外围设备的内部连接
PCI 时钟	33 MHz	根据PLB的时钟和寄存器设置衍生而来
MemClk	100 MHz	直接驱动SDRAM芯片，根据CPU时钟或通过寄存器设置进行配置衍生而来

讨论时钟设置通常必须和硬件设计同步进行。管脚的侦测选项根据处理器的应用来决定初始化的时钟配置。关于时钟的一些控制通常可以通过设置分频位实现。分频位通过处理器内部寄存器设置完成对时钟和子系统的控制。我们这里给出的基于405GP的例子中，最终的时钟配置通过管脚侦测和固件配置决定。设置初始的分频器和上电后立刻设置处理器寄存器位配置时钟选项，都是引导装入程序的职责。

C.3 SDRAM 设置

时钟配置完成以后，下一步是配置SDRAM控制器。处理器之间的控制器非常广泛，但是最终结果总是一样的：你必须提供正确的时钟和能够优化SDRAM子系统的时序。

如本书其他内容所述，掌握你配置的硬件的细节知识是最重要的，对于SDRAM更是如此。研究SDRAM的设计已经超出本附录的内容，但是有些基本内容你务必理解。很多厂商的SDRAM设备的数据手册提供了有用的技术说明，强烈建议你熟悉这些数据手册中的内容。要正确配置SDRAM子系统，硬件工程师不需要获得相关的学位，但是需要多下点功夫。

第7章介绍了U-Boot，这里我们查看一下配置U-Boot时，是如何配置405GP处理器上的SDRAM控制器的。回想第7章的一段代码，它取自4xx-specific目录下的start.S汇编文件，该文件是一段启示代码，是U-Boot中为SDRAM初始化提供的一个钩子。参考7.4.4节。代码清单C-1给出U-Boot中的.../cpu/ppc4xx/sdram.c file里sdram_init()函数。

代码清单C-1 U-Boot中的sdram_init()函数

```
01 void sdram_init(void)
02 {
03     ulong sdtrl;
04     ulong rtr;
05     int i;
06
07     /*
```

```

08      * Support for 100MHz and 133MHz SDRAM
09      */
10      if (get_bus_freq(0) > 100000000) {
11          /*
12           * 133 MHz SDRAM
13           */
14          sdtr1 = 0x01074015;
15          rtr = 0x07f00000;
16      } else {
17          /*
18           * default: 100 MHz SDRAM
19           */
20          sdtr1 = 0x0086400d;
21          rtr = 0x05f00000;
22      }
23
24      for (i=0; i<N_MB0CF; i++) {
25          /*
26           * Disable memory controller.
27           */
28          mtsdram0(mem_mcopt1, 0x00000000);
29
30          /*
31           * Set MB0CF for bank 0.
32           */
33          mtsdram0(mem_mb0cf, mb0cf[i].reg);
34          mtsdram0(mem_sdtr1, sdtr1);
35          mtsdram0(mem_rtr, rtr);
36
37          udelay(200);
38
39          /*
40           * Set memory controller options reg, MCOPT1.
41           * Set DC_EN to '1' and BRD_PRF to '01' for 16 byte PLB Burst
42           * read/prefetch.
43           */
44          mtsdram0(mem_mcopt1, 0x80800000);
45
46          udelay(10000);
47
48          if (get_ram_size(0, mb0cf[i].size) == mb0cf[i].size) {
49              /*
50               * OK, size detected -> all done
51               */
52              return;
53          }
54      }
55 }

```

第一个动作是读取405GP处理器的复用管脚，以决定SDRAM时钟的值。在本例中，我们可以看到两个推荐值：100MHz和133MHz。选择的常量将用在后面的函数中，以设置SDRAM控制器中适当的寄存器位。

从第24行开始，用一个循环为每达到5倍预定容量的内存设置参数。当然，U-Boot有一个逻辑用来支持4MB、16MB、32MB、64MB或128MB内存的一个组。这些容量在.../cpu/ppc4xx/sdram.c文件的表mb0cf中定义。根据405GP内存库配置寄存器需要的值，在表中用一个常量和这些内存容量关联在一起。循环如下：

```
for (i = each possible memory bank size, largest first) {
    select timing constant based on SDRAM clock speed;
    disable SDRAM memory controller;
    configure bank 0 with size[i], timing constants[i]
    re-enable SDRAM memory controller;

    run simple memory test to dynamically determine size;
    /* This is done using get_ram_size() */
    if ( tested size == configured size )
        done;
}
```

这个简单的逻辑会根据SDRAM时钟速度和U-Boot中硬件解码表中内存库容量的配置，向SDRAM控制器中插入正确的时间常量。通过这样的解释，你可以很容易联想到405GP参考手册中使用的库配置值。例如64MB DRAM，内存库控制寄存器设置如下：

Memory Bank 0 Control Register = 0x000a4001

PowerPC 405GP的用户手册中，所描述的内存库0控制寄存器如表C-2所示。

表C-2 405GP 内存库0-3配置寄存器字段

字 段	值	注 释
库地址 (BA)	0x00	该库的内存起始地址
容量 (SZ)	0x4	内存库的容量，本例中是64MB
地址模式 (AM)	0x2	决定内存的组织，包括行和列的位数。在本例中，模式2的含义是：行地址位为12，列地址位为9或者10，内部SDRAM库是4个。这些数据会在405GP的用户手册中以表格的方式提供
库使能 (BE)	0x1	这个库的使能位就由这个寄存器配置。405GP 中共有4个库配置寄存器

表中的值必须由设计人员根据板上使用的内存模块指定。

我们看一个时序的例子，加深理解一个典型的SDRAM控制器对时序的要求。假定一个SDRAM的时钟速度是100 MHz，容量是64 MB，那么代码清单C-1中的sdram_init()函数选择的时序常量如下：

```
SDRAM Timing Register    = 0x0086400d
Refresh Timing Register  = 0x05f00000
```

PowerPC 405GP的用户手册中，所描述的SDRAM时序寄存器如表C-3所示。

表C-3 405GP SDRAM时序寄存器字段

字 段	值	注 释
CAS潜伏期(CASL)	0x1	SDRAM CAS潜伏期, 这个值从SDRAM芯片规范中直接获得。它是芯片从发出读命令(CAS信号)那一刻到数据总线上的数据可用后的一段延时。在本例中, 0x1代表两个时钟周期, 从405GP的用户手册中可以查到
预充电到激活(PTA)	0x1	SDRAM的预充电(pnecharge)命令, 使给定的行无效。反之, 活动(activate)命令使一贯给定的行可以用于后期的访问, 比如在突发周期期间。这个时序参数将执行预充电到下一个活动周期的最小时间, 由SDRAM芯片规定。正确的数值必须通过SDRAM芯片规范获得。在本例中, 0x1代表两个时钟周期, 从405GP的用户手册中可以查到
读/写命令预充电最小周期(CTP)	0x2	这个时序参数执行SDRAM读(或写)命令到下一个预充电命令之间的一段延时。正确的数值必须通过SDRAM芯片规范获得。在本例中, 0x2代表三个时钟周期, 从405GP的用户手册中可以查到
SDRAM初始命令(LDF)	0x1	这个时序参数执行地址声明或者选择库周期的一段延时。正确的数值必须通过SDRAM芯片规范获得。在本例中, 0x1代表两个时钟周期, 从405GP的用户手册中可以查到

代码清单C-1中的U-Boot实例配置了最后的时序参数, 它将刷新时序寄存器的值。这个寄存器需要一个单独的字段决定SDRAM控制器必须的刷新间隔。这个字段表示间隔是根据SDRAM时钟频率算出来的。在这个实例中, 我们假定SDRAM的时钟频率是100 MHz, 为寄存器编写的值是0x05f0_0000。通过查看PowerPC 405GP的用户手册, 我们了解到它的刷新请求是15.2 μ s。和其他的时序参数一样, 这个值由SDRAM芯片规范指定。

一个典型的SDRAM芯片的每一行都需要一个刷新周期, 而每行都必须在厂商规定的最小时间内完成刷新。在“参考资源”中提到的芯片中, 厂商指定了8192个行必须在64ms内刷新完成, 这就需要每隔7.8ms产生一个刷新周期, 以满足这个特殊设备的规范。

C.4 小结

SDRAM设备相当复杂。这个附录只是介绍一个非常简单的例子, 以帮助你清楚SDRAM控制器设置的复杂程度。SDRAM控制器执行一个关键的函数, 它必须被正确地设置。除了钻研规范和消化这里提供的信息外, 没有其他好办法。这份附录给出的两个参考文档是极好的入门教程。

参考资源

AMCC 405GP 嵌入式处理器用户手册

AMCC 公司

www.amcc.com/Embedded/

Micron Technology, Inc.

同步DRAM MT48LC64M4A2 数据手册

<http://download.micron.com/pdf/datasheets/dram/sdram/256MSDRAM.pdf>

D.1 代码资料库和开发资料

网络上集中讨论Linux开发的几个地址，这里给出一些最重要的网络站点。

主要的内核源码树

www.kernel.org

主要的内核库

www.kernel.org/git

PowerPC相关的开发和邮件列表

<http://ozlabs.org/>

MIPS相关的开发

www.linux-mips.org

ARM-Linux相关的开发

www.arm.linux.org.uk

开源工程的主页

<http://sourceforge.net>

D.2 邮件列表

成百甚至上千的邮件列表会满足Linux和开源开发的每一个方面。这里要稍作考虑，在邮递这些列表以前，确信你自己适合这个列表的内容。

大多数列表保留着存档，并可进行搜索。这里是你应该首先要查询的。在绝大多数情况下，你的问题已经被提过而且也有了答案。从这里开始阅读，可以获得如何最佳使用公共邮件列表的建议：

Linux 内核邮件列表FAQ

www.tux.org/lkml

各种Linux内核相关的邮件列表

<http://vger.kernel.org>

Linux 内核邮件列表，内容很多，只包括内核开发

<http://vger.kernel.org/vger-lists.html#linux-kernel>

D.3 Linux 新闻和开发

很多新闻站点有时也值得浏览。这里列出一些较流行的：

LinuxDevices.com

www.linuxdevices.com

PowerPC新闻和其他信息

<http://penguinppc.org>

综合的Linux新闻和开发

www.lwn.net

D.4 开源观察和讨论

下面的公共站点包含了围绕开源法律问题的信息：

www.open-bar.org



附录 E

BDI-2000配置文件示例

```
; bdiGDB configuration file for the UEI PPC 5200 Board
; Revision 1.0
; Revision 1.1 (Added serial port setup)
; -----
; 4 MB Flash (Am29DL323)
; 128 MB Micron DDR DRAM
;
[INIT]
; init core register
WREG   MSR           0x00003002 ;MSR : FP,ME,RI
WM32   0x80000000 0x00008000 ;MBAR : internal registers at 0x80000000
        ; Default after RESET, MBAR sits at 0x80000000
        ; because it's POR value is 0x0000_8000 (!)

WSPR   311           0x80000000 ; MBAR : save internal register offset
        ; SPR311 is the MBAR in G2_LE

WSPR   279           0x80000000 ;SPRG7: save internal memory offsetReg: 279

; Init CDM (Clock Distribution Module)
; Hardware Reset config {
;   ppc_pll_cfg[0..4] = 01000b
;   XLB:Core -> 1:3
;   Core:f(VCO) -> 1:2
;   XLB:f(VCO) -> 1:6
;
;   xlb_clk_sel = 0 -> XLB_CLK=f(sys) / 4 = 132 MHz
;
;   sys_pll_cfg_1 = 0 -> NOP
;   sys_pll_cfg_0 = 0 -> f(sys) = 16x SYS_XTAL_IN = 528 MHz
; }
;
; CDM Configuration Register
WM32   0x8000020c 0x01000101
        ; enable DDR Mode
        ; ipb_clk_sel = 1 -> XLB_CLK / 2 (ipb_clk = 66 MHz)
        ; pci_clk_sel = 01 -> IPB_CLK/2

; CS0 Flash
```

```

WM32    0x80000004  0x0000ff00 ;CS0 start = 0xff000000 - Flash memory is on
CS0
WM32    0x80000008  0x0000ffff ;CS0 stop  = 0xffffffff

; IPBI Register and Wait State Enable
WM32    0x80000054  0x00050001 ;CSE: enable CS0, disable CSBOOT,
                                ;Wait state enable\
                                ; CS2 also enabled

WM32    0x80000300  0x00045d30 ;BOOT ctrl
                                ; bits 0-7: WaitP  (try 0xff)
                                ; bits 8-15: WaitX  (try 0xff)
                                ; bit 16: Multiplex or non-mux'ed (0x0 = non-muxed)
                                ; bit 17: reserved (Reset value = 0x1, keep it)
                                ; bit 18: Ack Active (0x0)
                                ; bit 19: CE (Enable) 0x1
                                ; bits 20-21: Address Size (0x11 = 25/6 bits)
                                ; bits 22:23: Data size field (0x01 = 16-bits)
                                ; bits 24:25: Bank bits (0x00)
                                ; bits 26-27: WaitType (0x11)
                                ; bits 28: Write Swap (0x0 = no swap)
                                ; bits 29: Read Swap (0x0 = no swap)
                                ; bit 30: Write Only (0x0 = read enable)
                                ; bit 31: Read Only (0x0 = write enable)

; CS2 Logic Registers
WM32    0x80000014  0x0000e00e
WM32    0x80000018  0x0000efff

; LEDs:
; LED1 - bits 0-7
; LED2 - bits 8-15
; LED3 - bits 16-23
; LED4 - bits 24-31
; off = 0x01
; on  = 0x02
; mm 0xe00e2030 0x02020202 1 (all on)
; mm 0xe00e2030 0x01020102 1 (2 on, 2 off)

WM32    0x80000308  0x00045b30 ; CS2 Configuration Register
                                ; bits 0-7: WaitP  (try 0xff)
                                ; bits 8-15: WaitX  (try 0xff)
                                ; bit 16: Multiplex or non-mux'ed (0x0 =
non-muxed)
                                ; bit 17: reserved (Reset value = 0x1, keep it)
                                ; bit 18: Ack Active (0x0)
                                ; bit 19: CE (Enable) 0x1
                                ; bits 20-21: Address Size (0x10 = 24 bits)
                                ; bits 22:23: Data size field (0x11 = 32-bits)
                                ; bits 24:25: Bank bits (0x00)
                                ; bits 26-27: WaitType (0x11)
                                ; bits 28: Write Swap (0x0 = no swap)
                                ; bits 29: Read Swap (0x0 = no swap)
                                ; bit 30: Write Only (0x0 = read enable)

```

```

; bit 31: Read Only (0x0 = write enable)

WM32 0x80000318 0x01000000 ; Master LPC Enable

;
; init SDRAM controller
;
; For the UEI PPC 5200 Board,
; Micron 46V32M16-75E (8 MEG x 16 x 4 banks)
; 64 MB per Chip, for a total of 128 MB
; arranged as a single "space" (i.e 1 CS)
; with the following configuration:
; 8 Mb x 16 x 4 banks
; Refresh count 8K
; Row addressing: 8K (A0..12) 13 bits
; Column addressing: 1K (A0..9) 10 bits
; Bank Addressing: 4 (BA0..1) 2 bits
; Key Timing Parameters: (-75E)
; Clockrate (CL=2) 133 MHz
; DO Window 2.5 ns
; Access Window: +/- 75 ns
; DQS - DQ Skew: +0.5 ns
; t(REFI): 7.8 us MAX
;
; Initialization Requirements (General Notes)
; The memory Mode/Extended Mode registers must be
; initialized during the system boot sequence. But before
; writing to the controller Mode register, the mode_en and
; cke bits in the Control register must be set to 1. After
; memory initialization is complete, the Control register
; mode_en bit should be cleared to prevent subsequent access
; to the controller Mode register.

; SDRAM init sequence
; 1) Setup and enable chip selects
; 2) Setup config registers
; 3) Setup TAP Delay

; Setup and enable SDRAM CS
WM32 0x80000034 0x0000001a ;SDRAM CS0, 128MB @ 0x00000000
WM32 0x80000038 0x08000000 ;SDRAM CS1, disabled @ 0x08000000

WM32 0x80000108 0x73722930 ;SDRAM Config 1 Samsung
; Assume CL=2
; bits 0-3: srd2rwp: in clocks (0x6)
; bits 507: swt2rwp: in clocks -> Data sheet suggests
; 0x3 for DDR (0x3)
; bits 8-11: rd_latency -> for DDR 0x7
; bits 13-15: act2rw -> 0x2
; bit 16: reserved
; bits 17-19: pre2act -> 0x02
; bits 20-23: ref2act -> 0x09
; bits 25-27: wr_latency -> for DDR 0x03
; bits 28-31: Reserved

```

```

WM32      0x8000010c  0x46770000 ;SDRAM Config 2 Samsung
                                ; bits 0-3: brd2rp -> for DDR 0x4
                                ; bits 4-7: bwt2rwp -> for DDR 0x6
                                ; bits 8-11: brd2wt -> 0x6
                                ; bits 12-15: burst_length -> 0x07 (bl - 1)
                                ; bits 13-16: Reserved

; Setup initial Tap delay
WM32 0x80000204  0x18000000 ; Start in the end of the range (24 = 0x18)
Samsung

WM32      0x80000104  0xf10f0f00 ;SDRAM Control (was 0xd14f0000)
                                ; bit 0: mode_en (1=write)
                                ; bit 1: cke (MEM_CLK_EN)
                                ; bit 2: ddr (DDR mode on)
                                ; bit 3: ref_en (Refresh enable)
                                ; bits 4-6: Reserved
                                ; bit 7: hi_addr (XLA[4:7] as row/col
                                ; must be set to '1' 'cuz we need 13 RA bits
                                ; for the Micron chip above
                                ; bit 8: reserved
                                ; bit 9: drive_rule - 0x0
                                ; bit 10-15: ref_interval, see UM 0x0f
                                ; bits 16-19: reserved
                                ; bits 20-23: dgs_oe[3:0] (not sure)
                                ; but I think this is req'd for DDR 0xf
                                ; bits 24-28: Resv'd
                                ; bit 29: 1 = soft refresh
                                ; bit 30 1 = soft_precharge
                                ; bit 31: reserved

WM32      0x80000104  0xf10f0f02 ;SDRAM Control: precharge all
WM32      0x80000104  0xf10f0f04 ;SDRAM Control: refresh
WM32      0x80000104  0xf10f0f04 ;SDRAM Control: refresh

WM32      0x80000100  0x018d0000 ; SDRAM Mode Samsung
                                ; bits 0-1: MEM_MBA - selects std or extended MODE reg 0x0
                                ; bits 2-13: MEM_MA (see DDR DRAM Data sheet)
                                ; bits 2-7: Operating Mode -> 0x0 = normal
                                ; bits 8-10: CAS Latency (CL) -> Set to CL=2 for
DDR (0x2)
                                ; bit 11: Burst Type: Sequential for PMC5200 ->
0x0
                                ; bits 12-14: Set to 8 for MPC5200 -> 0x3
                                ; bit 15: cmd = 1 for MODE REG WRITE

WM32      0x80000104  0x710f0f00 ;SDRAM Control: Lock Mode Register (was
0x514f0000)

; ***** Initialize the serial port *****
; Pin Configuration
WM32      0x80000b00  0x00008004 ; UART1

; Reset PSC
WM8       0x80002008  0x10          ; Reset - Select MR1

```

```

WM16  0x80002004  0      ; Clock Select Register - 0 enables both Rx &
Tx Clocks
WM32  0x80002040  0      ; SICR - UART Mode
WM8   0x80002000  0x13    ; Write MR1 (default after reset)
                        ; 8-bit, no parity
WM8   0x80002000  0x07    ; Write MR2 (after MR1) (one stop bit)

WM8   0x80002018  0x0     ; Counter/Timer Upper Reg (115.2KB)
WM8   0x8000201c  0x12    ; Counter/Timer Lower Reg (divider = 18)

; Reset and enable serial port Rx/Tx
WM8   0x80002008  0x20
WM8   0x80002008  0x30
WM8   0x80002008  0x05

;
; define maximal transfer size
TSZ4  0x80000000  0x80003FFF ;internal registers
;
; define the valid memory map
MMAP  0x00000000  0x07FFFFFF ;Memory range for SDRAM
MMAP  0xFF000000  0xFFFFFFFF ;ROM space
MMAP  0xE00E0000  0xE0EFFFFF ; PowerPC Logic
MMAP  0x80000000  0x8fffffff ; Default MBAR
MMAP  0xC0000000  0XCFFFFFFF ; Linux Kernal

[TARGET]
CPUTYPE 5200      ;the CPU type
JTAGCLOCK 0       ;use 16 MHz JTAG clock
WORKSPACE 0x80008000 ;workspace for fast download
WAKEUP 1000      ;give reset time to complete
STARTUP RESET
MEMDELAY 2000    ;additional memory access delay
BOOTADDR 0xfff00100
REGLIST ALL
BREAKMODE SOFT ; or HARD
POWERUP 1000
WAKEUP 500
MMU XLAT
PTBASE 0x000000f0

[HOST]
IP 192.168.1.9
FORMAT ELF
LOAD MANUAL ;load code MANUAL or AUTO after reset
PROMPT uei>

[FLASH]
CHIPTYPE AM29BX16 ;Flash type (AM29F | AM29BX8 | AM29BX16 | I28BX8 |
I28BX16)
CHIPSIZE 0x00400000 ;The size of one flash chip in bytes
BUSWIDTH 16 ;The width of the flash memory bus in bits (8 | 16 |
32)

```

```
WORKSPACE 0x80008000 ;workspace in internal SRAM
FILE      u-boot.bin
FORMAT    BIN 0xFFFF0000
ERASE     0xFFFF0000 ;erase a sector of flash
ERASE     0xFFFF1000 ;erase a sector of flash
ERASE     0xFFFF2000 ;erase a sector of flash
ERASE     0xFFFF3000 ;erase a sector of flash
ERASE     0xFFFF4000 ;erase a sector of flash

[REGS]
FILE      $reg5200.def
```

